Development of a Radiation Resistant Communication Node for Satellite Sub-Systems

by

Emile Jacobus Thesnaar

Thesis presented in fulfilment of the requirements for the degree of Master of Engineering (Research) in the Faculty of Engineering at Stellenbosch University

Supervisor:

Mr Arno Barnard

Department Electrical and Electronic Engineering

April 2014

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

20 February 2014

Abstract

Within a complex electronic system, sub-system communication forms the backbone of the functionality of any satellite. It allows multiple processors to run simultaneously and data to be shared amongst them. Without it, a single processor would have to control the entire satellite. Not only would such a design then be overly complicated, but the processor would also not have sufficient capacity to service all the components efficiently.

Furthermore the detrimental effects that radiation have on integrated circuits are well documented and can be anything from a single bit flip to a complete integrated circuit failure. If not repaired, a failure on a sub-system communication bus could lead to the loss of the entire satellite.

Die goal is to create more radiation resistant Controller-Area-Network (CAN) node. Since a full triple modular redundant design will have a large footprint and high power consumption, a combination of techniques will be applied and tested. The goal is to achieve improved footprint utilisation over triple modular redundancy, while still maintaining good resistance to Single Event Upsets (SEU).

By applying simulation, it was sufficiently proven that the implementation of the individual techniques used functioned according to expectations. These techniques included error detection and correction using Hamming Codes, single event transient filter and triple modular redundancy. Having applied these mitigation techniques, the footprint of the CAN controller increased by only 116%. Simulation showed that the Error Detection and Correction and Triple Modular Redundancy worked effectively with the CAN controller, and that the CAN controller could function as originally intended. Using radiation testing, the design proved to be more resistant to SEUs than the unmitigated CAN controller.

It was thus shown that through using a combination of mitigation techniques, it is possible to develop an optimal design with a high level of resistance against Single Event Upsets, utilizing a smaller footprint than implementing Triple Modular Redundancy.

Opsomming

Sub-stelsel kommunikasie vorm die basis van die funksionaliteit in 'n komplekse elektroniese stelsel soos 'n satelliet. Dit skep die vermoë om veelvoudige verwerkers gelyktydig te laat funksioneer en inligting tussen hulle te deel. Sonder sub-stelsel kommunikasie, sal 'n enkele verwerker die hele sateliet moet beheer. Dit sal nie net die hele ontwerp oorkompliseer nie, maar die verwerker sal ook nie genoeg kapisteit hê om al die komponente effektief te diens nie.

Die newe-effekte van bestraling op geïntegreerde stroombane is goed gedokumenteer en kan wissel van 'n enkele omgekeerde bis, tot die vernietiging van die geïntegreerde stroombaan. Indien die fout in die kommunikasiestelsel nie herstel word nie, kan dit lei tot die verlies van die hele sateliet.

Die doel is om 'n meer bestraling bestande Controller-Area-Network (CAN) nodus te skep. Aangesien 'n volle drie-dubbele-modulêre-oortollige ontwerp 'n baie groot area beslaan en hoë krag verbruik het, gaan 'n kombinasie van versagting tegnieke toegepas en ge-evalueer word. Die doel is om beter area benutting as die drie-dubble-modulêre-oortollige ontwerp te kry, terwyl 'n goeie weerstand teen foute behoue bly.

Deur middel van simulasies is voldoende bewyse gelewer dat die implimentasie van die individuele versagting tegnieke soos verwag funktioneer. Hierdie tegnieke sluit in, fout opsporing en regstelling deur middel van Hamming kodes, enkele geval oorgangs verskynsel filter asook drie-dubbele-modulêre-oortollige ontwerp. Nadat versagting meganismes toegepas is, het die area verbruik van die CAN beheerder toegeneem met slegs 116%. Simulasies het bewys dat Fout Opsporing en Regstelling en Drie-Dubbele-Modulêre-Oortollige ontwerp tegnieke binne die CAN beheerder korrek funktioneer, terwyl die CAN beheerder self funktioneer soos dit oorspronklik gefunksioneer het. Deur middel van bestralingstoetse, is dit bewys dat die ontwerp meer bestand is teen foute geïnduseer deur bestraling as die onbeskermde CAN beheerder.

Dit is dus bewys dat deur gebruik te maak van verskeie versagting tegnieke dit moontlik is om 'n optimale ontwerp te implimenteer, met 'n hoë weerstand teen foute, maar met 'n laer area verbruik as die van 'n Drie-dubbele-Modulêre-Oortollige ontwerp.

Acknowledgements

It would have been impossible to start, let alone complete, a project such as this without the support and assistance of a number of people who deserve my humblest thanks.

Niki Steenkamp for his insights during the design process.

Asic Design Services for the supply of the FPGA devices for the project, as well as their continued support during the project.

National Research Foundation for their support by providing the staff time and test facilities at iThemba LABS, Cape Town, to perform the radiation experiments.

Drs Rudolph Nchodu, Ricky Smit, Retief Neveling and Peane Maleka for their assistance at iThemba LABS prior and during radiation testing.

Arno Barnard for his continued guidance and assistance throughout the project.

My family for all their moral support during the project.

Most of all, Marna Rörich, who was always there for me. For her patience, continued support and loving care. Her moral and emotional support helped me immensely in making this project a success.

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations and Acronyms

ACF	Acceptance Filter
ACS	Attitude Control System
ADC	Analogue to Digital Converter
AHB	Advanced High-Performance Bus
APB	Advanced Peripheral Bus
ASB	Advanced System Bus
BGA	Ball Grid Array
BRP	Baud Rate Pre-scaler
BSP	Bit Stream Processor
BTL	Bit Timing Logic
CAN	Controller Area Network
CCD	Charge-coupled device
CMOS	Complementary Metal Oxide Semiconductor
CRC	Cyclic Redundancy Check
DUT	Device Under Test
DFF	D Flip Flop
DMA	Direct Memory Access
DMR	Dual Modular Redundancy
EDAC	Error Detection and Correction
EPS	Electronic Power System
ESA	European Space Agency
eV	Electron Volt

FIFO	First In First Out
FPGA	Field Programmable Gate-Array
GG	Guard Gate
GPIO	General Purpose Input Output
GPS	Global Positioning System
HDL	Hardware Descriptive Language
IBO	Bit Order Inverter
I^2C	Inter-Integrated Circuit
IC	Integrated Circuit
ID	Identification
IP	Intellectual Property
ISIS	Innovative Solutions In Space
JAXA	Japan Aerospace Exploration Agency
JTAG	Joint Test Action Group
LABS	Laboratory for Accelerator Based Sciences
LEO	Low Earth Orbiting
LVDS	Low Voltage Differential Signal
MAC	Media Access Control
MBU	Multiple Bit Upset
MIT	Massachusetts Institute of Technology
MOS	Metal Oxide Semiconductor
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
MRAM	Magneto-resistive Random Access Memory
MUX	Multiplexer
NASA	National Aeronautics and Space Administration
NVM	Non-Volatile Memory
OBC	On-board Computer
PCB	Printed Circuit Board
PHY	Physical Layer

PLL	Phase Locked Loop
PNPN	Positive-Negative-Positive-Negative
PSU	Power Supply Unit
QFP	Quad Flat Pack
RAM	Random Access Memory
RKA	Russian Federal Space Agency
SCL	Serial Clock Line
SDA	Serial Data line
SDRAM	Synchronous Dynamic Random Access Memory
SEB	Single Event Induced Burnout
SEE	Single Event Effect
SEFI	Single Event Functional Interrupt
SEGR	Single Event Gate Rupture
SEL	Single Event Latch-up
SET	Single Event Transient
SEU	Single Event Upset
SoC	System-on-Chip
SPI	Serial Peripheral Interface
SRAM	Synchronous Random Access Memory
SSC	Separated Sector Cyclotron
SSTL	Surrey Satellite Technology Ltd
TID	Total Ionizing Dose
TTL	Transistor-Transistor Logic
TMR	Triple Modular Redundancy
UART	Universal Asynchronous Receiver/Transmitter
UK	United Kingdom
VHDL	Very-High-Speed-Integrated-Circuit Hardware Descriptive Language
VHF	Very High Frequency
UHF	Ultra High Frequency
XML	Extensible Mark-up Language

Chapter 1

Introduction and Problem Description

As with most electronic systems, satellite systems are modular systems. Generally satellites contain, among others, an Electrical Power System (EPS), an Attitude Control System (ACS), a Communication System and an On-board Computer (OBC). To simplify the design, the satellite is broken down into functional sub-systems which can easily be developed and tested.

In order for the satellite to function effectively, all the subsystems need to function in harmony. While each subsystem has its own functionality, it is the sole function of the OBC to command and oversee all the subsystems.

To communicate with the various subsystems, a communication bus is required. With Sunspace's Sumbandila satellite [7], Surrey Satellite Technology Ltd's (SSTL) SSTL-300 satellite platform [8] and Massachusetts Institute of Technology's (MIT) F-12 flight computer [9], the communication bus comes in the form of the Controller Area Network (CAN) bus. For cubesats, most of the components listed on CubeSatShop [10], including the Innovative Solutions In Space (ISIS) Very High Frequency (VHF) Downlink / Ultra High Frequency (UHF) Uplink Full Duplex Transceiver [11], CubeComputer [12] and CubeSense [13], list Inter-Integrated Circuit (I^2C) as the applied bus interface.

When a satellite leaves the safety of the Earth's atmosphere, it is subject to the damaging radiation from both the sun, and other unknown cosmic entities. The radiation can have a damaging effect on the electronic components in the satellite and even lead to the failure of the mission. Sumbandila was launched on 17 September 2009 and suffered a failure to the EPS and one of the Charge-Coupled Device (CCD) camera boards between the launch and 30 March 2010. This led to the loss of control over the satellite, and the loss of three of the six spectral bands used for imaging [14].

In the event of a failure in one of the communication controllers, it could lead to the failure of the communication bus. Without the communication bus, the OBC would not be able to command the sub-systems, and the sub-systems will not be able to report back.

The communication bus is therefore a single point of failure on-board a satellite. Although mitigation techniques could be applied to avoid a communication bus failure, it would still

be possible for some sub-systems to lock the communication bus.

The goal of this project is to protect a communication node to be more resistant against the effects of radiation. This will be done by analysing and implementing various mitigation techniques on a communication controller. In order to implement these mitigation techniques, a HDL implementation of a communication controller will be used. After implementing the mitigation, the footprint usage should be less than that of Triple Mitigation Redundancy (TMR).

By making sure that the communication node is more resistant to radiation induced upsets, the point of failure could be moved away from the communication bus. This thesis will start by studying literature on radiation, sub-system communication, Field Programmable Gate Arrays (FPGAs) and upsets in FPGAs in Chapter 2. In Chapter 3, the design of a mitigated CAN controller will be discussed, along with the development of a testing platform to test the CAN controller's resistance to upsets. The design and population of the Printed Circuit Board (PCB) will be discussed in Chapter 4, followed by the alterations made to the CAN controller. In Chapter 5 the results from simulations and radiation testing will be discussed. Conclusions will be made in Chapter 6 and further possible work will also be covered.

Chapter 2

Literature Study

In this chapter a short background will be given of the sources and effects of radiation along with ways to protect against them. Sub-system communication in large designs will also be discussed, followed by a discussion on the different types of FPGAs. Finally, a background on different methods of System on Chip (SoC) communication will be given. To finish this chapter, the simulation and testing of Single Event Effects (SEEs) and Single Event Upsets (SEUs) will be discussed.

2.1 Radiation

While the sources and effects of radiation are well documented, a short background on the subject will be given, followed by a discussion on known protection against the effects.

2.1.1 Sources of Radiation

The earth's atmosphere not only serves life forms with breathable air, but it also acts as a giant screen to filter out the dangerous radiation from the hostile space environment, along with most of the infrared light [15]. Infrared light forms part of a wide band of energies, called the electromagnetic spectrum [16]. This spectrum ranges from low energy radio and microwave radiation to high energy X-rays and gamma rays.

Cosmic rays originate from beyond our solar system. These high energy particles consist of mainly heavy ions and atomic nuclei [17]. The exact origin of cosmic rays is still unknown, and data from the Fermi Space Telescope suggests that it originates from super novae [18]. The energy of these particles can reach up to 3×10^{20} electron-Volt (eV), and holds a real threat to microelectronics outside the protection of the earth's atmosphere and magnetic field [19].

Solar events refer to radiation originating from the sun. This includes the periodic increase and decrease in the solar wind, as well as the occasional solar flare. The sun is constantly emitting not only light, but also a stream of charged particles, or plasma. This stream of particles radiates outwards from the sun, and due to the rotation of the sun, forms waves referred to as solar wind. Solar wind consists mostly of electrons and protons, with energies

exceeding 100 MeV [20]. While not as dangerous as cosmic rays, consistent bombardment with these particles can also induce failures in microelectronics.

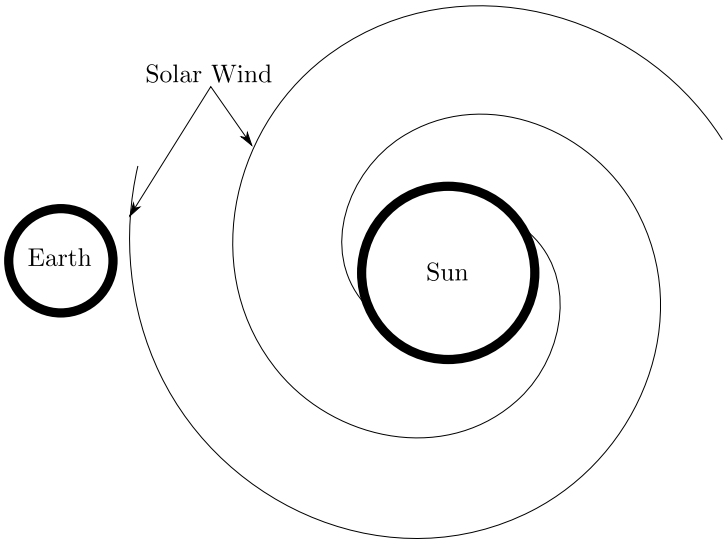

Figure 2.1 – Diagram Illustrating the Spiral Shape of the Solar Wind Due to the Rotation of the Sun.

Solar flares occur near sunspots when stored magnetic energy is suddenly released from the corona. These flares appear as arches on the sun's surface and release electrons, ions and atoms outwards. The particles have been recorded to reach energies of up to 1 MeV and even higher [21].

The Van Allen belts are belts of radiation in the upper magnetosphere of the earth. They are kept in place by the earth's magnetic field, and consist of the trapped particles emitted by the solar wind with the same energy of 100 MeV. These belts are located between the earth's surface and 40 000 km above the earth's surface. While most Low Earth Orbiting (LEO) satellites orbit below these belts, Geostationary Orbit (GEO) satellites are located well inside of these belts at roughly 36 000km [1]. The Global Positioning System (GPS) satellites are located between the two belts, at a height of roughly 7 600 km above the earth's surface.

Figure 2.2 shows Van Allen belts symmetrical around the magnetic axis of the earth rather than the rotational axis. The upper radiation belt is located roughly 19 000 km above the earth's surface. It contains mainly electrons and protons, with energies up to 100 MeV, which are captured by the earth's magnetosphere. The inner radiation belt is located roughly 4 000 km above the earth's surface. This belt contains mainly high energy protons [1]. The energy of the electrons in the outer belt can go up to several thousand eV, while the protons can reach energies exceeding 100 MeV, depending of the energy of the protons in solar wind. Figure 2.3 shows the radiation dosages at various heights above the earth's surface due to the Van Allen belts [1].

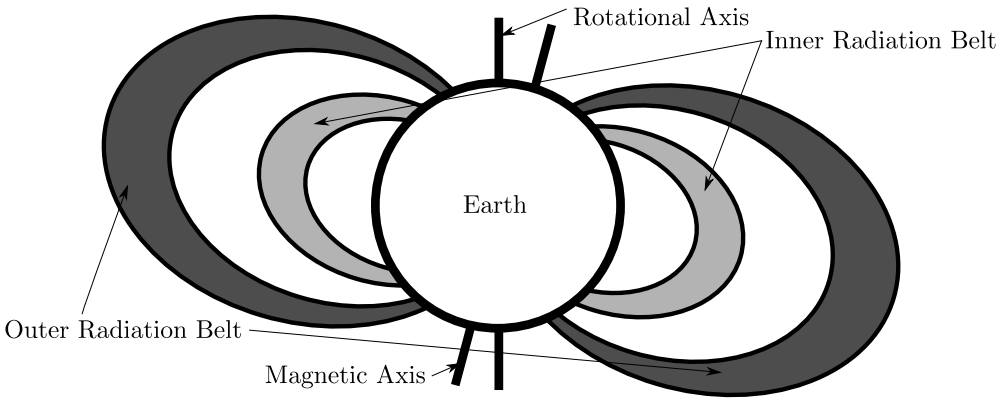

Figure 2.2 – Illustration of Van Allen Belts Surrounding Earth.

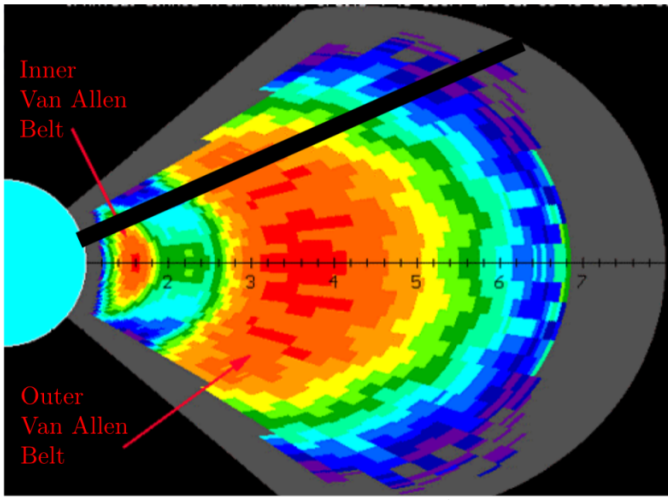

Figure 2.3 – Radiation Dosages Due to Van Allen Belts. Adapted From [1].

2.1.2 The Effects of Radiation

The effects of radiation on electronic components manifest itself in many ways, from a temporary signal spike to total device failure.

2.1.2.1 Lattice Displacement

Lattice displacement is the displacement of atoms in the atomic structure of the Integrated Circuit (IC). The high energy particle loses energy as it penetrates the semiconducting layer inside the IC. Through Rutherford scattering and Compton interactions, the particle is slowed as it transfers energy to particles in the lattice [22]. When the energy of the particle is high enough, the impact with other particles can result in a particle being displaced from its original location to form lattice vacancies, to replace another particle in the lattice or to lodge into an empty space in the lattice. With the alteration in the electron-hole pairs, the

excess carriers can deposit a charge with undesired consequences [22].

Lattice displacement is most commonly caused by high energy particles with energies above 100 MeV. These particles include alpha, beta and gamma particles, protons, neutrons, electrons and other heavy ions. Often, high temperature or annealing allows the recombination of displaced atoms with empty locations in the lattice [23].

2.1.2.2 Ionization Effects

Ionization effects are caused when the lattice is hit by either high or low energy charged particles. Electrons are added or removed from a neutral or partially ionized atom. Changing the electron count of the atom results in the bond between the atoms of the molecule breaking down, changing the chemical composition of the material [23].

2.1.2.3 Total Ionizing Dose (TID) Effects

Prolonged exposure to radiation leads to a build-up of ionization effects and damage to the crystal lattice. The build-up of trapped charges in the Metal Oxide Semiconductor (MOS) transistor's silicon dioxide insulating layer escapes through tunnelling effects [24]. The tunnelling effect occurs when ions escape the insulating layer, generating a minute current.

In Complementary Metal Oxide Semiconductor (CMOS) logic, radiation exposure leads to the generation of electron-hole pairs in the insulating layers. When these pairs recombine, a minute current is generated [23]. In the event of a hole being trapped in the lattice structure, the bias of the gate gets altered, along with the threshold voltage [24].

2.1.2.4 Single Event Effects

In the event of the deposited charge being large enough, three types of Single Event Upsets (SEUs) can occur, namely transient, permanent or static.

Transient errors occur when the excess carriers deposit a charge resulting in an asynchronous signal propagating through the circuit. This spurious signal can either be latched into a memory element, or be filtered out by dominant signals inside the circuit [22].

Permanent errors are destructive and irreversible leading to physical damage of the circuit. Examples of these errors are Single Event Induced Burnout (SEB) and Single Event Gate Rupture (SEGR), which mainly occur in power transistors [22]. In parasitic Positive-Negative-Positive-Negative (PNPN) structured devices, the ionization track of a high energy particle is able to turn on both internal transistor structures. This results in the device creating a low resistance path between the upper and lower voltage rails, resulting in a high current short circuit. This is known as a Single Event Latch-Up (SEL). If the fault is noticed early enough, a power cycle can remove the SEL, preventing damage to the circuit.

Static errors occur when transients are latched into a memory element, or when a volatile memory element is struck by a higher energy particle, changing its value. Both SEUs and Multiple Bit Upsets (MBUs) can occur inside a circuit, but can be repaired through an

external interaction [22]. In the event of an SEU changing the control circuits leading to a change in device state, the SEU becomes a Single Event Functional Interrupt (SEFI). These errors can typically be corrected though a power cycle of the circuit, or through an external interaction.

2.1.3 Radiation Hardening

The term radiation hardening is used to describe the process through which a design is made more resistant against the effects of radiation. Several techniques exist to protect circuits from the effects of radiation, consisting of physically protecting the circuit or building redundancies into the circuit.

2.1.3.1 Hardening by Design

Integrated Circuits (ICs) are produced by doping, or adding impurities, to a semiconductor wafer, usually made from silicon. To create a hardened IC, the IC is produced using insulating substrates, such as silicon dioxide or sapphire, instead of the silicon wafer. While normal ICs can take a dose of between 5 and 20 krad, radiation hardened ICs can take a dose of several factors higher. The 4000 series logic (RadHard) can tolerate a dose of roughly 100 to 1000 krad [25].

Another method of protecting an IC against radiation is to use a substrate with a high band gap. The band gap is the energy gap between the valence band and the conduction band. In insulators, the conduction band has a higher energy than the valence band. This means that external energy is required to relocate electrons from the valance band to the conduction band. When a particle with high enough energy collides with the insulating substrate, it can create a conducting path through the insulator. To increase the band gap, substrates such as silicon carbide or gallium nitrate can be used which have roughly three times higher band gap than silicon [26].

Covering the IC in depleted boron can also reduce the effects of radiation inside the IC [27]. Depleted boron, or Boron-10, is used in Boron Neutron Capture Therapy for treating cancer patients. Boron-10 is attached to tumours using a tumour seeking compound, where-after it is irradiated using low energy neutrons. Boron-10 then disintegrates by absorbing the neutrons, and emits a gamma ray, an alpha particle and a lithium ion. These destroy the cancerous cells, leaving normal cells relatively unaffected [28].

Instead of using Synchronous Random Access Memory (SRAM) or Synchronous Dynamic Random Access Memory (SDRAM), the use of Magneto-resistive Random Access Memory (MRAM) is suggested as an alternative [29]. MRAM has been shown to be inherently radiation resistant [29]. While the construction and operation of MRAM are not covered by this project, the principle on which MRAM is based is that it does not store an electrical charge. Instead, it uses magnetic components to store information.

2.1.3.2 Dual and Triple Modular Redundancy

Duplicating a design is one of the most common methods of adding redundancy to a system. During the development of the submarine command system for the United Kingdom (UK) ministry of defence in 1983, it was decided that each central node should be duplicated to create a fault tolerant system. These nodes were interconnected using dual fibre optic local area networks [30].

While it is possible to detect differences between the two modules, selecting the correct signal is problematic. Modern Dual Modular Redundancy (DMR) designs use self-voting techniques to determine which signal should be used [31].

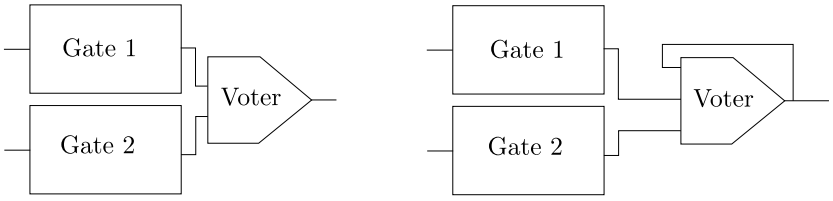

Figure 2.4 – Schematic Representation of a Traditional DMR Circuit and a Modern Implementation of a Self-Voting DMR Circuit

TMR is more resistant to upsets than DMR, but it is also more inefficient. TMR triplicates the design and uses majority voters on the outputs, thus increasing the area usage by more than three times. Similar to DMR, TMR can be applied on different levels of the design.

The first option is to triplicate the entire design on IC level. This is known as triple device redundancy. It involves placing three equivalent components and using the controlling logic as a majority voter. An example of this is can be found in commercial and military aircraft, where certain parts of the control system are triplicated to avoid malfunctions.

Having three identical ICs on the circuit board has its drawbacks. Not only does it require more than three times the area, but three times the weight is added and three times the power is consumed. In a satellite environment where area, weight and power budgets are limited, implementing TMR should be avoided.

The second option is found typically internal to an IC. This involves triplicating parts of the device, which on their own have no useful function, but in terms of their usage are crucial to the device. An example of this is triplicating the control registers on a processor.

Finally, individual low level gates can be triplicated. This method is the most complex and costly in terms of area usage. The area usage will be at least four times that of the original design, as each gate requires a majority voter.

2.1.3.3 Single Event Transient (SET) Filter

The SET filter is a mitigation technique used to filter out SET pulses using a delay element and comparison technique to suppress the effect of the pulse [32; 3; 2].

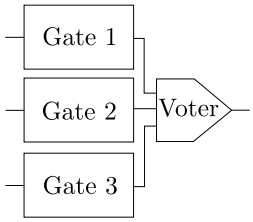

Figure 2.5 – Schematic Representation of a Traditional TMR Circuit

Originally, the comparison was made using a guard-gate (GG) [2], where one input is connected to the original source and the second is connected to the output of the delay unit. The GG consists of four Metal Oxide Semiconductor Field Effect Transistor (MOSFETs), with connected drain to source as shown in Figure 2.6. Inputs are connected to either the upper two or the lower two MOSFETs, while the output is connected to the connection between the two upper and lower two MOSFETs. Two inputs of the same value will result in two of the four MOSFETs turning on, pulling the output either high or low. Opposite inputs will result in a floating output, as the output is neither connected to a high or low.

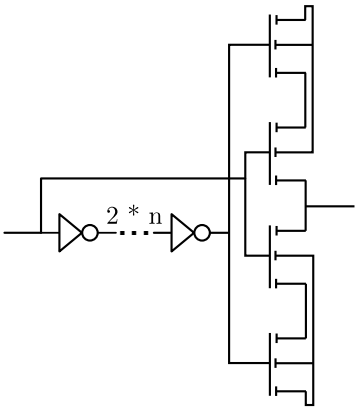

Figure 2.6 – Diagram of the SET Filter using GG as in [2]

While this is an elegant solution, it is practically flawed because a floating output will result in undesired behaviour in following sequential logic. Also, implementing a GG in the fabric of the FPGA is impossible due to the architecture of most FPGAs.

A new comparison technique was developed by Farouk Smith, which involves feedback from the output to do the voting [3]. Similar to previous filtering techniques, the delay is made by a set of equal number logic inverters, while the comparison is done using an AND-gate, OR-gate and a 2-port Multiplexer (MUX), illustrated in Figure 2.7 [3].

To explain the inner working of the SET filter, separate stages of the filter will be discussed below. A logic '0' to either of the inputs of the AND-gate, will result in the output being a logic '0'. If a SET pulse of logic '1' was to occur on the input, it will be shifted slightly behind the original pulse by the delay element on the second input to the AND-gate. If the SET pulse is shorter than the delay, the pulses on the inputs will not overlap, resulting in

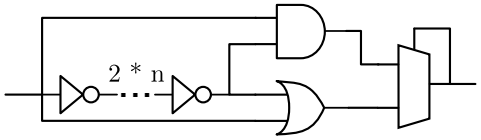

Figure 2.7 – Diagram of the SET filter as Proposed in [3]

the output remaining a logic '0'. If the SET pulse duration is longer than the delay, the two pulses will overlap and a short pulse will propagate though the AND gate. In Figure 2.8, the top signal represents the case where the pulse is shorter than the delay element, while the bottom signal is longer.

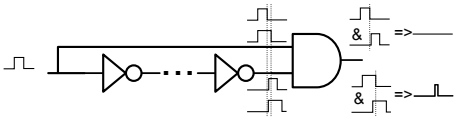

Figure 2.8 – Illustration of the SET filter Implementation's AND-gate Logic

Opposite to the AND-gate, a logic '1' on either of the inputs of the OR-gate, will result in the output being a logic '1'. Following the same argument as above, the output of the OR-gate will remain a logic '1' if the SET pulse is shorter than the delay. In Figure 2.9, the top signal represents the case where the pulse is shorter than the delay element, while the bottom signal is longer.

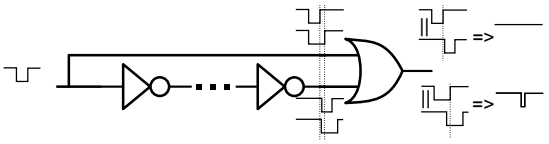

Figure 2.9 – Illustration of the SET filter Implementation's OR-gate Logic

If the input to the SET filter is a logic '0', both the AND-gate and the OR-gate will output a logic '0'. The same can be said for a logic '1'. If no SET pulses are present, both inputs of the MUX will be the same, and therefore the output will be the same as the input. Assuming the input was originally a logic '1', a SET pulse occurs and it is shorter than the delay, the AND-gate will output two logic '0' pulses, one for the original pulse and one for the delayed pulse. Meanwhile the OR-gate will remain at a logic '1'. Because the output was a logic '1' before the SET pulse occurred, the MUX is selected to use the input from the OR-gate, which remained at a logic '1'. If the SET pulse was longer than the delay, the AND-gate will output a logic '0' from the start of the SET pulse until the end of the delayed SET pulse. The OR-gate will output a logic '0' for the overlapping part of the original and delayed SET pulse. When this happens, the output of the MUX changes to a logic '0', selecting the AND gate as input. Since the AND-gate remains at a logic '0' until the delayed SET pulse is passed, the full SET pulse will propagate through the SET filter.

Similarly, if the input was a logic '0' originally, and a logic '1' SET pulse occurs which is shorter than the delay, the MUX would be set to use the AND-gate, which will remain at a logic '0'. If a SET pulse longer than the delay occurs, the AND-gate will be a logic '1' for the overlapping part which will select the OR-gate as input, resulting in the SET pulse propagating through the filter.

In the event of the SET pulse being shorter than the delay of the delay element, the SET filter will theoretically filter out all SET pulses. A recent study on flash-based FPGAs showed that for the Actel ProAsic3, the maximum SET pulse width was roughly 3-4 nanoseconds [33]. The SET filter's delay element should therefore have a delay of 5 ns or more.

2.1.3.4 Error Detection and Correction (EDAC) using Hamming Code

Error detection on data can be done by storing extra information about the data. This is normally done using parity bits or Cyclic Redundancy Check (CRC) bits. Identifying the wrong bit and correcting it could also be accomplished, given that enough extra information has been stored. One of these techniques is the Hamming Code.

Hamming code is a method of detecting and correcting single bit errors within data. The Hamming code encoding and decoding algorithm scales well over different lengths of data. A study by IBM showed that 98% of errors are single bit errors [34], and therefore the Hamming Code would suffice for use in this project. Since the CAN controller uses 8 bit registers, the following content was adapted to reflect input data of 8 bits, protected with a 4 bit Hamming code [35].

Definition of **data bits**: The data being encoded

Definition of **parity bits**: Bits inserted between data bits to form the code word.

Definition of **code word**: Data bits and parity bits combined

To create a code word, the original data is padded with extra bits. These bits are located at positions 2^x where $x = 0, 1 \ldots n$. If the data byte to be encoded is [d_0 d_1 d_2 d_3 d_4 d_5 d_6 d_7] the Hamming code is represented by [p_0 p_1 d_0 p_2 d_1 d_2 d_3 p_3 d_5 d_0 d_6 d_7], where d_x represents the original data bits and p_x represents the inserted parity bits [36; 37].

The encoded bits are allocated by the relationship between the data bits. The value of p_x is given by checking the parity of selected bits. When the parity is even, the parity bit is set to 0. When the parity is odd, the parity bit is set to 1. This results in the parity of the data bit and parity bit always being even. The relationship between the parity and data bits are given by Equations 2.1.1 to 2.1.4.

$$p_0 = parity\{d_0, d_1, d_3, d_4, d_6\} \tag{2.1.1}$$

$$p_1 = parity\{d_0, d_2, d_3, d_5, d_6\} \tag{2.1.2}$$

$$p_2 = parity\{d_1, d_2, d_3, d_7\} \tag{2.1.3}$$

$$p_3 = parity\{d_4, d_5, d_6, d_7\} \tag{2.1.4}$$

Once the parity bits are inserted, the parity of the data bits, along with the parity bit, will always be even. This relationship is the basis of the Hamming code.

To calculate which bit is incorrect, the parity of the selected data bits and parity bit is recalculated using Equations 2.1.5 to 2.1.8. If the parity is returned as odd, it indicates a problem with that group. By calculating the parity for each group, the incorrect position can be calculated using Equation 2.1.9.

$$e_0 = parity\{p_0, d_0, d_1, d_3, d_4, d_6\} \tag{2.1.5}$$

$$e_1 = parity\{p_1, d_0, d_2, d_3, d_5, d_6\} \tag{2.1.6}$$

$$e_2 = parity\{p_2, d_1, d_2, d_3, d_7\} \tag{2.1.7}$$

$$e_3 = parity\{p_3, d_4, d_5, d_6, d_7\} \tag{2.1.8}$$

$$Position = \sum_{x=0}^{n} e_x \cdot 2^x \tag{2.1.9}$$

Using the position given by the previous equation, the incorrect bit can be corrected. The position of the faulty bit can be either a data bit, or a parity bit.

These mitigation techniques formed the basis from which a custom mitigation solution was developed. During a later stage, the application of each technique will be discussed.

2.1.4 Different Units

Four units are used throughout the thesis when referring to radiation namely: electronvolt, rad, gray (Gy) and Ampere (A). Electronvolt is used describe the energy of a particle, more specifically, the amount of energy gained or lost by moving a single electron over an electric potential difference of one volt. One electronvolt is equal to $1.6x10^{-19}$ joule (J) of energy.

Rad is used to describe the absorbed radiation dose and can be defined as 0.01 J/Kg of mass, or $6.25x10^7$ MeV/kg Rad is deprecated unit of measurement and has since been replaced with gray, which is the equivalent to 100 rad, or $6.25x10^9$ MeV/kg.

Finally Ampere is used to measure the amount of particles travelling though a certain cross section of the conductor. In the case of radiation, these particles can be heavy-ions, protons or neutrons. Depending on the absorptivity of the materials inside the IC, the amount of particles passing through the IC and the energy of the particle, the total dose, or rad, can be calculated.

2.2 Sub-System Communication

Communication between all sub-systems in any multi-system design is key to the functionality of the complete system. Several different communication protocols exist, but only one will be used.

2.2.1 Commonly Used Communication Systems

Although several network communication standards are available, only a few are employed in the space industry.

2.2.1.1 SpaceWire

SpaceWire is a communication network developed specifically for the use in satellites. It was developed by the European Space Agency (ESA), in collaboration with National Aeronautics and Space Administration (NASA), Japan Aerospace Exploration Agency (JAXA) and Russian Federal Space Agency (RKA) [38]. Nodes are connected though a point to point serial connection, with optional switching routers.

SpaceWire can reach speeds up to 200 Mbps [38]. It utilizes Low-Voltage Differential Signalling (LVDS) across a 9 pin connection [39]. Using data strobe encoding, it allows for easy clock recovery and jitter tolerance. Data strobe encoding specifies that either the data signal or strobe signal should change once every clock cycle. To recover the clock, the data and strobe signals are channelled through an XOR gate.

While SpaceWire is capable of high speeds, it's usage is limited to communication between two devices without a router. This makes it impractical for use as a shared sub-system communication bus.

2.2.1.2 Universal Asynchronous Receiver/Transmitter (UART)

UART converts parallel data into a serial data stream for point-to-point communication. There are several communication standards which are used on top of UART, such as RS-232, RS-422 and RS-485. While all these standards have different specifications in the way communication takes place, all of them are based on UART.

UART consists of a transmission line and a receiving line. The transmission line from one device is connected to the receiving line of another. This allows stable communication between two devices with speeds up to 115.2 kbps. At higher frequencies, the slew rate will start to approach the 4% maximum of bit period allowed. Due to the simple nature of UART, both devices need to be set to exactly the same data rate to avoid corruption of data [40].

Similar to SpaceWire, using UART to create a bus is impractical.

2.2.1.3 Serial Peripheral Interface (SPI)

SPI bus as it is called by Motorola, or Microwire trade marked by National Semiconductor, is a full duplex serial communication bus [41]. Due to industry acceptance, SPI is currently one of the dominant communication standards for peripheral communication. SPI is driven by a single master at a time, with the number of slaves limited to the number of General Purpose Inputs and Outputs (GPIO) available on the master. It is also possible for inter-processor communication using SPI.

SPI can reach a clock frequency of up to 50 MHz reliably, based on components currently available. Higher frequencies are possible, but are limited by the production quality of the PCB and components. It uses Transistor-Transistor Logic (TTL) over a 3-wire connection, with an extra wire for each slave. Two single directional data wires are used for data transmission and is synchronised using a third clock signal. For each slave device present, an extra chip select wire is added to the interface.

SPI works well for communication between a controller and the peripherals attached to it. To implement an SPI bus between controllers would seem impractical, as there always needs to be a master controller. Having a single component in control of the bus could lead to the loss of all sub-system communication, should it fail.

2.2.1.4 I2C

I^2C is a communication bus developed by Phillips in 1982 to connect low speed devices to microcontrollers [42]. Several competitors have since released products compatible with Phillips' I^2C system, which led to the reduction of licensing fees on the patent. A fee is still payable in order to get a formal address for a device from Phillips. Several revisions of I^2C have since been released, the latest being version 4.0, which is able to reach speeds of up to 5MHz [42].

I^2C consists of a Serial Data Line (SDA) and Serial Clock Line (SCL), which are both implemented using open-drain circuitry. These lines are pulled up to the voltage level used by the microcontroller using resistors, and pulled low by the I^2C controller when transmitting a logic '0'.

Communication takes place between two devices, where one acts as the master and the other acts as the slave. The master generates the clock signal for the SCL line and sends a start bit to the bus. The master then writes the address on the SDA line, and then expects an acknowledgement from the slave. The final bit of the address specifies whether a read or write transaction will follow. During a write transaction, the data is written to the bus by the master, and then waits for an acknowledgement from the slave. For a read transaction, the slave writes the data to the bus and sends an acknowledgement to the master. When the transaction is completed, the master will send a stop bit, releasing the bus. The number of data bytes written can continue until the sender decides to release the bus.

In the event of two I^2C controllers starting transmission at the same time, each controller will check whether it has control over the bus or not. Since each I^2C controller can only drive the bus down to a logic '0', the first controller to detect a logic '0' on the bus when transmitting a logic '1' will stop transmission, as it has lost control over the bus.

2.2.1.5 CAN

CAN is a communication bus developed for use inside of motor vehicles. The goal was to develop a communication bus which allows several sub-systems to communicate without a single host. It was developed by Robert Bosch GmbH in 1983, and released in 1986 [43].

CAN is a message based communication standard where every node can transmit and receive messages with up to eight bytes of data. CAN 2.0 A specifies that each message contains an 11 bit Identification (ID) which is used to determine which CAN controller should accept the message. The ID is also used to arbitrate which CAN controller has control over the bus. CAN 2.0 B offers an extended mode where the IDs of 29 bits can be transmitted.

Bus activity can be represented by either a recessive state, or logic '1', or a dominant state, or logic '0'. In the recessive state, the bus is not driven by any logic, as opposed to the dominant state where the transceiver is pulling the bus low. In the event of two CAN controllers transmitting at the same time, the first to transmit a recessive state, loses arbitration and backs off from transmitting.

To compensate for phase shifts between different sub systems, each bit is divided into four segments. The first segment is used to indicate a sync period. If there is a mismatch between the bus and the CAN controller, the CAN controller will delay extra cycles in the second segment to make up for the shift in phase. The last two segments are used to determine the duration of each bit, as well as where, relative to the end of the bit transmission, the bit should be sampled.

At a range of 40 meters, the maximum speed of CAN is roughly 1 Mbps. At higher rates reflections in the cables start to distort the signal, making it harder to determine the correct bus value.

2.2.1.6 Comparison of Sub-System Communication Methods

While most of the aforementioned communication standards have their advantages and disadvantages, each of them has a unique application. Table 2.1 lists a summary of all the communication networks.

For communication between two devices, either SpaceWire or UART could be used, UART being the most simplistic, offering slow speeds and no form of data protection. SpaceWire can reach speeds up to 400 times greater than UART. Using data strobing allows SpaceWire to work across different domains by making the clock easily available. Unfortunately, neither of these have the potential to form a practical communication bus.

SPI can also be considered a point-to-point communication bus, as the number of connections increases as the number of peripherals increases. If the chip selects are taken away, SPI boils down to be similar to both SpaceWire and UART. With SPI, the shared data lines and higher speeds make it an almost viable solution. Unfortunately, having a single master in control of the bus, results in a single point of failure which could bring down all sub-system communication.

The only two viable options remaining are I^2C and CAN. Both these buses allow the connection of multiple devices with multiple masters. While I^2C is capable of reaching 4 times the speed of CAN, I^2C messages are directed to a single device. With CAN, a message can be sent to multiple devices.

Communication	Wires	Type	Speed	Pros	Cons
SpaceWire	9	P-P	200 Mbps	Fast, noise immune (LVDS), jitter resistant, clock recovery	Point to point, extra logic to convert to bus
UART	2	P-P	115.2 kbps	Simple	Slow, point to point, clock sensitive
SPI	3+n	Bus	50 MHz	Fast, bus	Master dependant, bus width grows as nodes grow, node failure can cause bus failure
I^2C	2	Bus	5 MHz	Fast, bus, arbitration	Node failure can cause bus failure
CAN	2	Bus	1 Mbps	Noise immune (LVDS), arbitration, CRC, phase shift protection	Requires some form or processor, requires transceiver

Table 2.1 – Summary of different sub-system communication networks

Both I^2C and CAN make use of arbitration to avoid bus contention. Arbitration is the process through which bus nodes determine which has control over the bus. Since I^2C controllers can only drive the bus down to a logic '0', the first controller which reads a logic '0' on the bus when transmitting a logic '1', has lost arbitration. With CAN, arbitration is done using the ID, allowing more important message to pass through.

With I^2C, any number of data bytes can be transferred to and from a single address. With CAN, the data bytes are limited to eight. Where CAN gains the advantage, is the added protection of CRC bits on each message.

In summary:

- CAN transmits messages to multiple devices, were I^2C transmits to a single device.
- CAN has built in CRC to confirm message data, I^2C does not.
- CAN compensates for phase shifts by resynchronizing at each bit.
- The differential CAN signals are immune to noise.

From the above mentioned reasons, it is clear that CAN is the bus of choice for use subsystem communication. It has already been adopted into the automotive industry as the de facto communication bus and is already used on many satellites.

2.2.2 CAN in Hardware

As with any novel device, the CAN controller was initially implemented on a single device, which could be added to almost any design. With interfaces varying from SPI to 8051 bus, it could connect to most microcontrollers. These devices could function separate from other hardware, and in the event of failure, be switched off. Protecting these devices, either involves placing multiple buses on the design, or changing the design of the device completely. To change the design access to Intellectual Property (IP) would be required and the cost involved of having a device manufactured is more than the budget of this project.

With the advances in the construction of integrated circuits, devices such as the CAN controller can now be implemented directly on the microcontroller itself.

While both the above implementations add CAN to the sub-module of a project, protecting these implementations from the effects of radiation is costly, not only footprint wise, but financially as well.

The only way to protect these implementations is through DMR or TMR, increasing the area usage on the PCB which is problematic for both the power and weight budget of a satellite project. Another option is to improve the design of the chip itself, replacing the substrate with a more resistant one. Not only will this cost be prohibitive, but most vendors will probably not allow access to the design of the die.

An alternative would be to implement the CAN controller in an FPGA, where the design can be customized to fit the budgets of the project, while adding radiation resistance to the design.

2.2.3 SJA1000 CAN Controller from OpenCores.org

Igor Mohor, a developer at OpenCores.org, has already implemented the SJA1000 CAN controller in Verilog. This project comes complete with a test bench to validate the working of most of the internal parts of the CAN controller. The Verilog implementation was also verified using the Bosch Very high speed Hardware Descriptive Language (VHDL) reference system to be CAN compliant.

The internal structure of the CAN controller is customizable using configuration files in which different interfaces can be specified, First-In-First-Out (FIFO) memory implementation and a memory diagnostic stream. The memory diagnostic stream is used when the FIFO is implemented in actual Random Access Memory (RAM), rather than in the fabric of the FPGA.

2.3 FPGAs

There are several suppliers of FPGAs, each using different technologies to implement the FPGA circuitry. Each of the technologies and suppliers have their strong and weak points, and choosing the right one is crucial to the design.

2.3.1 Different Types of FPGA

There are two main types of reconfigurable FPGAs namely, SRAM and flash based FPGAs.

SRAM based FPGAs are based on static memory. These need to be reprogrammed by an external boot device on power-up. These devices are normally faster than flash based FPGAs, while also using less power. Unfortunately, the memory in which the configuration is stored is susceptible to upsets from radiation. If such an upset occurs, the device needs to be reprogrammed resulting in system downtime. Apart from the FPGA which is susceptible to upsets, the circuitry required to program the FPGA is also susceptible, therefore adding extra complication to any design.

Flash based FPGAs store their configuration in flash memory. As opposed to SRAM based FPGAs, these are live at power-up and require no re-programming. The flash memory storing the configuration is resistant to upsets, as per interview with Niki Steenkamp and [44]. A recent radiation test using the same device as chosen in the design has shown that the configuration is not totally immune to upsets [45]. The downside is that the Flash memory uses more power.

There are also FPGAs that are one-time programmable. These are based on antifuse technology. As opposed to a fuse, antifuse starts with a high resistance, which turns into a short circuit if the voltage across it rises higher than the threshold. The problem with this fixed configuration is that it is not re-programmable. These are not practical during a research project, but are useful in satellites where reprogramming is not an issue.

2.3.2 Actel FPGA Comparison

Actel is currently the only manufacturer developing flash based FPGAs. They offer a wide range of FPGAs, each with its own application [46].

2.3.2.1 Igloo

The Igloo range of Actel devices is known for their low power consumption, low cost and small footprint. The Igloo nano range has the lowest power usage and smallest size FPGA available of all the Actel devices. With the new Igloo2 range of FPGAs released by Microsemi, the exact details surrounding the device operation is not yet known. The Igloo Plus is a low power FPGA with enhanced input-output functionality. Finally, the Igloo E range of FPGAs has the same low power consumption but with a larger die size. The Igloo range also offers small amounts of internal SRAM and Non-Volatile Memory (NVM). The Igloo E also offers a Cortex M1 enabled version of the FPGA.

The Igloo range of FPGAs is a candidate device for the space environment. While the low power usage and small IC footprint is ideal for both power and weight budgets, the small gate count could be problematic. For a device with a high enough gate count, the footprint will be limited to a Ball Grid Array (BGA). As will be discussed later, the BGA footprint is not ideal for usage in space, or for use on projects that require debugging.

2.3.2.2 ProASIC3

The ProASCI3 FPGAs are typically larger and use more power, but also offer higher operating speeds than the Igloo range of FPGAs. The ProAsic3 range offers embedded dual-port SRAM and FIFO blocks, advanced I/O and high speed operation. The ProAsic3L device offers low power consumption compared to the other ProAsic3 devices. The ProAsic3 nano range is smaller than the other ProAsic3 devices in both gate count and footprint. The ProAsic3E devices are larger than other devices and require more power. Both the ProAsic3 E and L devices offer a Cortex-M1 enabled version of the FPGA [47].

The ProAsic3 range of FPGAs uses roughly 200 times more power than the Igloo range, but offers faster operation. A study has also shown that where the ProAsic3 has SET pulses of up to 4 nano seconds, the Igloo range can have SET pulses of up to 9 nano seconds [33].

2.3.2.3 Fusion

The Fusion range of FPGAs combines analogue blocks, large flash memory blocks and clock management circuitry on a single device. The aim of the Fusion range of FPGAs is to implement an entire system-on-chip[48].

In 2010 Actel, now Microsemi, also created a SmartFusion range of FPGAs which differs from other FPGAs. Instead of having one large programmable FPGA fabric, the SmartFusion has a similar structure to a microcontroller. The SmartFusion range of FPGAs offers peripherals such as Ethernet Media Access Control (MAC), I^2C controllers, SPI controllers, Analogue to Digital Converters (ADCs) and Digital to Analogue Converters (DACs). During the design process, certain blocks are activated and routed similarly to the FPGA fabric. These blocks can interface with a smaller FPGA fabric where the user can develop his own hardware [49].

2.4 SoC Intercommunication

SOC intercommunication refers to the communication between components, more specifically, between a microprocessor and peripherals.

2.4.1 Wishbone Interconnect

The Wishbone Interconnect was developed to reduce system-on-chip integration issues involving the connection of IP cores. The objective was to create a common interface for all IP cores which was portable and reusable. Wishbone is not copyrighted and can be used royalty-free in any design, making it the common interface when developing custom IP cores [50].

The Wishbone Interconnect offers two types of connections namely, point-to-point and bus connection. Point-to-point is used when connecting two IP cores directly to one another, while the bus connection requires a bus management system to connect multiple IP cores [51]. A list of the Wishbone Interconnect signals is listed in Table 2.2.

The Wishbone Interconnect is flexible and can be customized for the user's needs while still maintaining a robust standard. The well documented specification makes it easy to develop a Wishbone interface for an IP Core, allowing rapid development of IP Cores and SoC designs.

2.4.2 8051

The 8051 interface used on the SJA1000 CAN controller is not a formal bus definition; instead it is an adaptation of the internal bus interface used to communicate with internal memory. The internal bus specifies separate data, address and control buses which are used for data transfer. In the Intel 8051 each data transfer takes 6 clock cycles, with the address loaded on the third cycle and data read or written on the fifth and sixth cycles [52].

Adapting the above 8051 interface to communicate with external devices resulted in the 8051 bus was formed. The external 8051 bus consists of a shared 8-bit address and data bus, with several control signals.

When a read or write operation is initiated, the master loads the address on the *PORT_I/O*

Signal Name	Purpose
CLK_I	Clock Signal
RST_I	Reset Signal
TGD_I/O	Tag Bus Signal (Unused in most designs)
DATA_I/O	Data bus of width 8, 16, 24 or 32 bits
ACK_I/O	Acknowledges completion of bus cycle
ADR_I/O	Address bus of width 2 to 32 bits
CYC_I/O	Indicates a valid bus cycle is in progress
STALL_I/O	Indicates slave is not able to accept a transfer
ERR_I/O	Indicates invalid bus cycle termination
LOCK_I/O	Indicates current bus cycle is uninterruptible
SEL_I/O	Device select signals
RTY_I/O	Indicates interface is busy and cycle should be retried
STB_I/O	Indicates a valid bus cycle
TGA_I/O	Address tag signal
TGC_I/O	Cycle tag signal
WE_I/O	Write enable signal

Table 2.2 – List of Signals Used in the Wishbone Interconnect [6]

Signal Name	Purpose
RST_I	Device Reset Signal
CLK_I	Device Clock Signal
ALE_I	Address Latch Enable Signal
RD_I	Read Signal
WE_I	Write Signal
PORT_I/O	Data and Address Bus
CS_I	Chip Select Signal

Table 2.3 – List of Signals Used in the 8051 Bus

bus, and asserts the *ALE_I* signal. Assertion refers to driving a signal to a logic '0'. The slave devices then clock in the address into its register. Depending on the read or write operation, the master then puts the *PORT_I/O* bus either in a high impedance state, or loads the data onto the bus. At this time, either the *RD_I* or the *WE_I* is asserted.

This interface can be used on many different microcontrollers due to its simplicity, but due to the address latching process the processor is slowed down while interfacing with other devices.

2.4.3 AMBA

AMBA is an interface developed and patented by ARM Limited. The AMBA protocol is an open standard which is used in the SoC development. When it was first developed in 1996, it consisted only of the Advanced System Bus (ASB) [53]. The ASB was used to connect multiple processors and controllers on the same bus, while slower peripherals were connected to the Advanced Peripheral Bus (APB). In revision 2 of the AMBA interface, the AMBA High-Performance Bus (AHB) was introduced which replaced the ASB [54]. The main change was the increase in speed of the AHB bus, which was now a single-edge protocol [54]. This meant that interactions with the bus could take place in a single clock cycle.

Although AMBA has now reached revision 4.0, revision 2.0 will be applied in this project. Microsemi's Libero software uses both AMBA 2 and 3 in their designs, but for simplicity it was decided to use AMBA 2.

The AHB bus can consist of multiple masters, multiple slaves, an arbiter and a decoder. To ensure that only one master has control of the bus, the arbiter blocks all interactions from other masters, once a master has initiated a transaction. To select which slave the master is communicating with, the decoder decodes the address and selects the correct device. The AHB bus is mainly used to connect high speed devices such as microcontrollers, memory interfaces and Direct Memory Access (DMA) devices.

The APB bus consists of all the lower speed peripherals, and connects to the AHB via a AHB-to-APB bridge. While memory and high speed controllers normally have a wide addressing range, smaller and slower peripherals have a much narrower range. This allows a single APB bus to be connected to a single address range on the AHB bus, with multiple peripherals connected to the APB bus.

Both the AHB and APB buses consist of an address, data-in and data-out buses, as well as several control signals. The address signal is automatically decoded to enable the slave device, while a single write-not-read signal allows for the selection of read or write operations.

2.5 Simulation of SEEs

To simulate a Hardware Descriptive Language (HDL) file, a simulator is required. There are several simulation packages available, such as: Verilog-XL, NCVerilog, VCS, Aldec, Finsim, Icarus Verilog, ISE Simulator and Modelsim. Modelsim is bundled with Microsemi Libero software and supports all Actel devices. During the design of a project using the Libero

software, the user can simulate the design at three stages in the design process: pre-synthesis, post-synthesis and post-layout.

Pre-synthesis simulation simulates the basic functionality of the design, without taking any delays into account. This is handy to confirm the logic behind the design. Post-synthesis simulation adds an estimated delay to each gate generated by the synthesis of the design. This helps to confirm time-critical functionality when delays are introduced to the system. Post-layout adds more accurate delays to each gate, as well as delays in the routing of internal connections. Post-layout simulation is the most accurate model of the final implementation in hardware.

The simulation of SEEs involves injecting transients or upsets into the circuit during simulation. For SETs, this involves changing a wire's value for a few hundred pico seconds. For SEUs, the content of a register needs to be changed. When simulating SEEs, several questions should be posed: What is the failure rate prediction and quantification? What is the recovery time on failure? What is the difficulty of recovery? What is the difficulty of system validation after mitigation [55]?

2.6 SEU Testing Using a Particle Beam

Radiation testing will be done at iThemba Laboratory for Accelerator Based Sciences (LABS) using the A-Line scatter chamber as shown in Figure 2.10. The Separated Sector Cyclotron (SSC) used at iThemba LABS is the most powerful accelerator in the southern hemisphere [4]. It is used for proton and neutron therapy, as well as isotope production and various other beams for nuclear physics research. The SSC is used to accelerate protons to an energy of 200 MeV on Mondays and Fridays, and 66 MeV during the remainder of the week [56].

Radiation testing involves the generation of high energy particles, such as heavy ions, protons and neutrons, in a controlled environment. This is accomplished using the SSC which accelerates the particles while under vacuum to increase the energy of the particles. Since the SCS functions under vacuum, all the connecting channels and testing chambers should be under vacuum. For this reason, the A-Line scatter chamber will be under vacuum when testing for this project is done. The particles are then guided towards the Device Under Test (DUT) using quadruple electromagnets. While protons and electrons are used for TID testing, with Co-60 and X-Rays as the preferred irradiation source [57], protons and heavy ions are used for SEE testing [58].

The number of generated particles passing through the DUT is measured in Ampere, and is usually in the 1×10^{-9} A range. Since most protons are charged, using Coulomb's constant, the number of particles can be calculated. The current, when using a charged particle beam, is measured through current measurement, but when neutral particles are used the capturing sensors, like scintillation detectors, are used to count the particles, after which the current can be calculated. During testing a proton beam will be used. For a proton beam, the

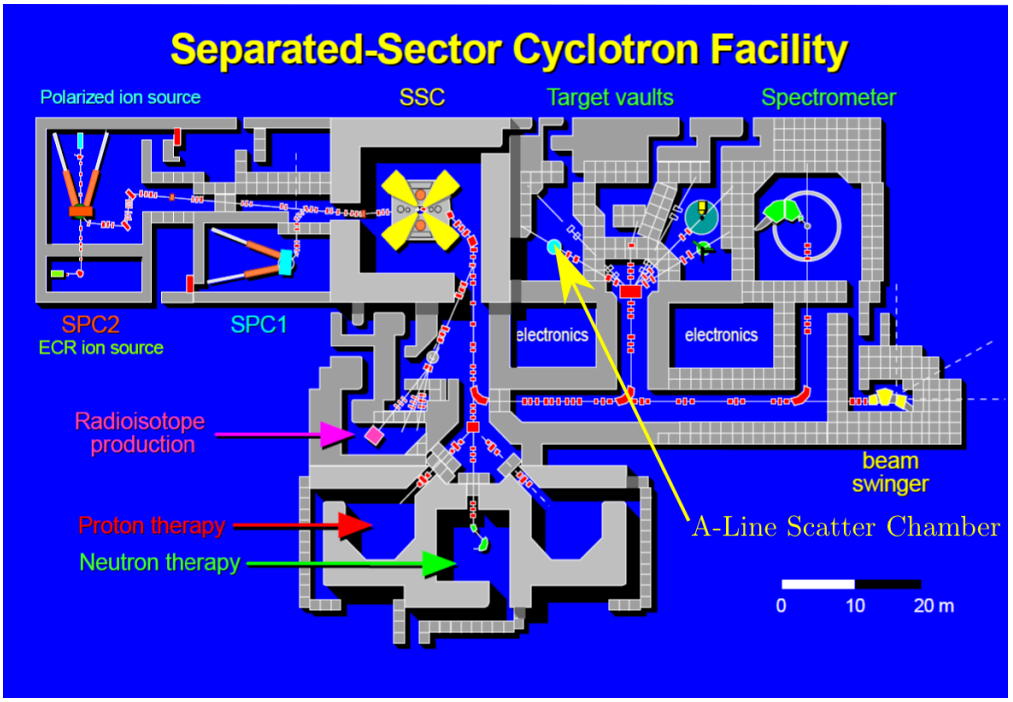

Figure 2.10 – Floor plan of iThemba LABS' Separated-Sector Cyclotron Facility [4]

current is measured and the particle count can be calculated. For a beam of 1nA, a total of $6,242 \times 10^9$ particles pass through the DUT each second as given by Equation 2.6.1.

$$N = \frac{I}{Q} = \frac{1 \times 10^{-9}}{1.60217657 \times 10^{-19}} = 6,242 \times 10^9 \tag{2.6.1}$$

At iThemba LABS, the beam cross-section is roughly 1 mm x 1 mm prior to the entry point into the vacuum chamber, and can be defocussed (or focussed) using electro magnets. Depending on the size of the die inside the DUT, the beam size should be changed to cover the entire die.

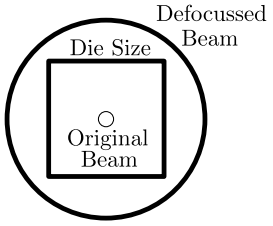

Figure 2.11 – Illustration of Beam Defocussing

While a total of $6,242 \times 10^9$ particles are entering the vacuum chamber, not all of these particles will pass through the die. Some of these particles will miss the die because of the difference in area between the die and the beam. Depending on the quality of the electro magnets used in the facility, the shape of the defocussed beam can also vary.

Chapter 3

Design

In this chapter, the partially successful attempt at translating the Verilog implementation of the CAN controller to VHDL will be discussed. Following on, the CAN controller will be dissected into functional units and analysed. The hardware required to test the CAN controller will be designed. An overview of the changes to the CAN controller will be given, followed by the design of the mitigation schemes. Finally, the final implementation and testing of the design will be covered.

3.1 CAN controller Verilog to VHDL translation

The HDL implementation of the SJA1000 CAN controller was written in Verilog. Although it is similar to VHDL, differences in the language result in different implementations. An attempt was made to translate the CAN controller from Verilog to VHDL.

3.1.1 Motivation for Translation

Both Verilog and VHDL have their advantages and disadvantages, and which to use is a personal preference. At the beginning of the project, the author was conversant with VHDL only. It was desirable to obtain a VHDL implementation of a CAN controller, but unfortunately most are protected by IP or are not completely functional. Currently, most FPGA designs in this research group are based on VHDL.

After researching, a Verilog implementation of the SJA1000 CAN controller was found on OpenCores.org. A VHDL translation of this CAN controller was also found in the LEON Project by Gaisler Research. The LEON project is an open source project using the LEON SPARC processor. The goal of the LEON project is to create a customizable VHDL processor with sub-components. This design will also be radiation resistant and SEU free [59]. It was decided to use this translation, and continue from there.

3.1.2 Problems

After analysing the files from the LEON project, the translation seemed to be intact. However, when attempting to use the VHDL version in a test-bench, the behaviour differed from

the Verilog version.

After closer inspection, it was found that several of the files had blocks of code that were missing. Comparing it to the Verilog source revealed that all the missing blocks were surrounded by *IFDEF* statements. *IFDEF* statements are used in Verilog to customize source code to be compiled using predefined constants. Whereas this functionality is available in Verilog, it is however not available in VHDL. Unlike Verilog, VHDL does not have a precompiler which deals with the IFDEF statements. VHDL has a similar function, called *generate*, which can generate components according to parameters. Unfortunately, it does not have the same power as the *IFDEF* in Verilog.

Not knowing the full extent of the damage to the translation, it was decided to rather start from scratch and since the VHDL does not have the *IFDEF* functionality, the customizability of the Verilog version had to be sacrificed to translate it. This will result in the CAN controller having a fixed interface, with the FIFO being implemented in the fabric of the FPGA, rather than using internal or external SRAM.

During the translation, files were constantly compared to the Verilog version. Starting with the smaller register files, the translation seemed easy enough as comparing both functionality and the synthesis results seemed to match the Verilog version. But when it came to bigger files such as the *can_bsp*, there were minor differences in the number of gates it used. There seemed to be radical changes in design of the resulting circuit from Synthesis. The differences in the resulting circuit raised concerns about internal differences, even though both Verilog and VHDL functioned similarly.

Upon completion of the translation, there was roughly a 4% logic cell usage difference on the 5500 logic cell design, with the resulting circuits differing functionally. These differences were considered to be minor and simply the effect of the difference between the Verilog and VHDL compilers.

Further testing of the translation again delivered undesirable results. When comparing a specific block translation from Verilog to VHDL, the functionality and resulting circuit matched. However, the bigger the file being translated, the bigger the differences in the resulting circuit.

Finally, the translation was abandoned due to large discrepancies between the translations which can be ascribed to the differences in the compilers. Attempts were made to use some of Synopsis' compiler directives to stop all optimizations, but with no success.

While the translation failed, much was learned from the translations. Translating each file revealed extensive detail about the inner-workings and construction of the CAN controller, as well as the differences and pros & cons of Verilog and VHDL.

3.2 SJA1000 CAN controller Breakdown

To understand the working of the SJA1000 CAN controller, it will be dissected down into several small components, each with an important role in the functionality of the CAN controller.

3.2.1 Visual Breakdown

Figure 3.1 shows the functional breakdown of the CAN controller. The Bit Timing Logic (BTL) interacts with the scaled clock and CAN bus, under control of the Bit Stream Processor (BSP). The BSP uses data from the control registers and reports the status back to the controlling device, while controlling the BTL.

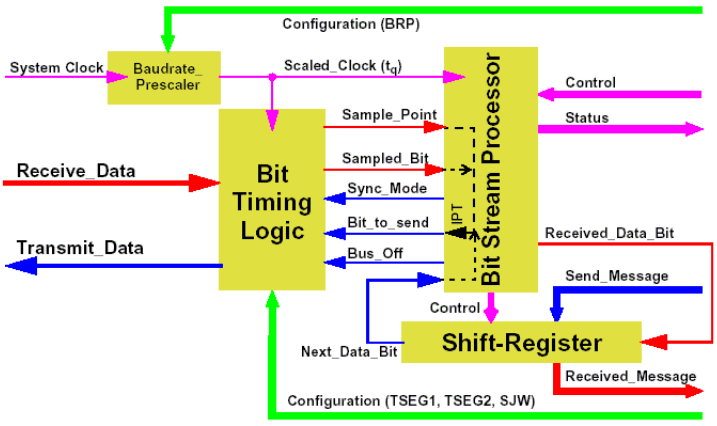

Figure 3.1 – Functional Breakdown of the SJA1000 CAN Controller [5]

Figure 3.2 shows the hierarchical design of the Verilog files of the CAN controller.

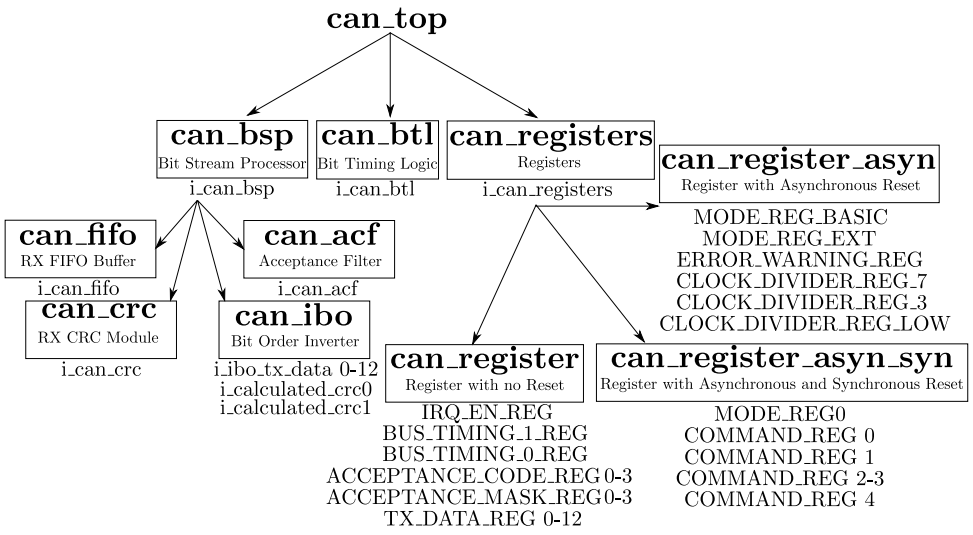

Figure 3.2 – HDL Structural Design of the SJA1000 CAN Controller

The *can_top* file is the top level file of the CAN controller. This file declares the interface between the CAN controller and the microprocessor. The *can_top* block is responsible for latching in the address and data before writing or reading to or from the internal components. It is also responsible for selecting when to read from the internal FIFO buffer or registers. Finally, it also initiates all the sub-components.

All the configuration registers are implemented in the *can_registers* block. There are three types of registers: registers without reset, registers with an asynchronous reset and registers with both asynchronous and synchronous resets.

According to the design, registers like the acceptance code and mask, transmit buffer, interrupt enable and bus timing data should not change on reset. These registers are all instances of *can_register*.

Registers such as the error warning register, mode register and clock divider registers, should reset to fixed values in the event of a reset. This is to reset the CAN controller into a known state when the reset occurred. These registers are all instances of *can_register_asyn*.

The command register and the least significant bit of the mode register need to be changed not only on an incoming reset, but also due to certain conditions in the CAN controller. The command register should be cleared as soon as the command is executed, and in the event of an error, the mode register should reset the CAN controller into a reset mode. To do this, *can_register_asyn_syn*. is used.

The three types of registers are all generic register files which require one or two parameters when they are instantiated: The width specifies the number of bytes required by the register, and the Reset Value specifies the value to which the register should reset.

Not only does *can_registers* house all the registers, it also generates an external clock for lower clock devices, generates an interrupt signal from signals coming in from other subcomponents and multiplexes data when a read is requested.

The *can_btl* block contains the bit timing logic from Figure 3.1. The timing logic scales down the input clock to the value required when sampling data by the Baud Rate Pre-scaler (BRP). Equation 3.2.1 is used to scale down the clock frequency to the appropriate bus frequency.

$$f_{scl} = \frac{f_{CLK}}{2(BRP + 1)} \tag{3.2.1}$$

Each bit of a CAN message consists of three timing sections. Firstly, there is a synchronization segment, which is used to synchronize phase shifts between CAN controllers. The phase shift is the result of unmatched clock signals from different areas of the CAN network. The second is timing segment 1. This is the delay until the bit should be sampled. It is followed by timing segment 2 which is the time lapsed until the end of the current bit transmission. The timing of these three sections are calculated using Equation 3.2.2 to 3.2.4

$$t_{SYNCSEG} = \frac{1}{f_{scl}} \tag{3.2.2}$$

$$t_{TSEG1} = \frac{TSEG1 + 1}{f_{scl}} \tag{3.2.3}$$

$$t_{TSEG2} = \frac{TSEG2 + 1}{f_{scl}} \tag{3.2.4}$$

The baud rate of the CAN bus can be calculated by adding these sections together.

$$f_{bus} = \frac{1}{t_{SYNCSEG} + t_{TSEG1} + t_{TSEG2}} \tag{3.2.5}$$

$$f_{bus} = \frac{f_{CLK}}{(3 + TSEG1 + TSEG2) \cdot 2 \cdot (BRP + 1)} \tag{3.2.6}$$

For a clock input of 20 MHz and a desired bus frequency of 125 kbps, the BRP value should be 7, TSEG1 = 2 and TSEG2 = 5. Refer to Equation 3.2.7

$$f_{bus} = \frac{20MHz}{(3 + 2 + 5) \cdot 2 \cdot (7 + 1)} = 125kbps \tag{3.2.7}$$

Together with the above information and input states from the bit stream processor, the *can_btl* samples bits from the CAN bus, and sends them to the *can_bsp* for processing.

The *can_bsp* block contains the bit stream processor seen in Figure 3.1. The bit stream processor uses the output from the bit timing logic in combination with an internal state machine, to process the bit stream. The state machine is used to determine which bit to sample next in the can message, as well as which bit to send next from the transmit buffer.

The *can_bsp* block also initiates the acceptance filter, which is used to determine whether an incoming message is meant for the specific controller. The *can_acf* block contains sequential logic to determine which filtering technique to use, and generates an *id_ok* signal to allow the message to be received or ignored.

The *can_fifo* block is also initiated in the BSP. The FIFO buffer initialises the memory required to implement the FIFO buffer. The user has the option to choose between device specific RAM modules or implementing the FIFO in FPGA fabric. Using RAM modules reduces the required FPGA fabric and can increase the maximum frequency of the device.

The *can_bsp* block manages data flow between itself, *can_top* and *can_fifo*, as well as generating all the errors and error counters for user debugging.

The interconnection of the sub-components forms a rather complex nest of wires not easily representable on paper. When implementing mitigation most of these connections will be severed to allow the required mitigation logic to be inserted.

3.2.2 Registers and Sequential Logic

After synthesis, the CAN controller used a total of 1 497 D-flip-flops and 3 694 sequential logic elements. Using Table 3.1, the exact usage of each component can be calculated. Depending on the nature of each component, the appropriate mitigation techniques could be inserted.

From the breakdown in Table 3.1, each sub-component can be analysed in terms of complexity, logic types and susceptibility to SEEs.

Module name and unit count	Register count	Logic Count	I/O Count	Max Freq
can_top	1 497	3 694	19	55.8 MHz
1x *can_bsp*	1 190	2 821	287	57 MHz
1x *can_acf*	1	195	123	NA
1x *can_crc*	15	37	19	177.4 MHz
1x *can_fifo*	879	1 251	37	72.7 MHz
3x *can_ibo*	0	0	16	NA
1x *can_btl*	30	155	36	101.0 MHz
1x *can_registers*	242	660	291	117.4 MHz
200x *can_register*	1	0	4	NA
20x *can_register_asyn*	1	0	5	NA
6x *can_register_asyn_syn*	1	2	6	NA

Table 3.1 – Breakdown of FPGA Usage of the Original CAN Controller Post-Synthesis

1. *can_acf*

   The Acceptance Filter (ACF) contains only 1 register to latch the *id_ok* signal. The high logic count makes it susceptible to SETs that could be converted to SEUs if latched into the register.

2. *can_crc*

   The CRC module is relatively small, but performs a crucial function. The low register and logic count makes it susceptible to both SETs and SEUs.

3. *can_fifo*

   The FIFO contains most of the registers in the design as it used to store incoming data the FPGA fabric. The FIFO's high count of both registers and logic units makes it susceptible to both SEUs and SETs.

4. *can_ibo*

   The Bit Order Inverters (IBO) used contains no logic units and is used as routing block to rotate the bit order from 0 - 8 to 8 - 0. It does not pose any risk to SETs and SEUs.

5. *can_bsp*

   The BSP itself contains a few registers with mainly sequential logic. Most logic units of any of the components are contained herein and is susceptible to both SEUs and SETs.

6. *can_btl*

   The BTL consists of a few registers and some sequential logic. It is susceptible to both SEUs and SETs.

7. *can_registers*

   The registers module consists of mostly sequential logic with the registers being located in individual files. The registers themselves are susceptible to SEUs. The sequential

logic is mainly used to generate the write enable signals for registers and is susceptible to SETs only.

3.3 Hardware Design

To test the resistance to SEEs of the design, it had to be programmed into an FPGA and then be radiated. While there are development kits available, the layout and construction of these kits make them less than ideal for the use in radiation testing. For testing to deliver useful data, the FPGA containing the design needed to be isolated from other hardware. A custom testing PCB had to be developed that suited the needs of this project.

A fellow student, Fancois Nolte, also worked on a similar project. His goal was to implement a radiation resistant soft-core processor in an FPGA, applying similar mitigation techniques to those in this project. To save cost and development time on both projects, a hardware collaboration was suggested. The distribution of the workload is discussed in Appendix A.

3.3.1 Hardware Requirements

Before the hardware design process could commence, all hardware requirements had to be listed.

3.3.1.1 Device Under Test

The DUT is an FPGA which will be radiated using a particle beam to simulate the space radiation environment. It should support both designs, but not simultaneously.

After synthesis of the CAN controller, it used roughly 5 200 logic units of FPGA fabric. To implement full TMR would typically triple the device usage, and add extra overhead for the voting logic. Therefore, an FPGA with a minimum of 16 000 logic units should be enough.

Francois's design consisted of an OpenRisc soft-core processor, as found at OpenCores.org. Unmitigated, it consisted of roughly 15 000 logic units. While a full TMR would be ideal, it would not be viable as all components needed to be triplicated, including memory. Instead, a limit of 35 000 logic units was put on his design.

Production capabilities also had to be kept in mind when performing device selection. While Ball Grid Array (BGA) devices have a small footprint, debugging them using probes can be difficult as opposed to Quad Flat Pack (QFP) devices. When soldering BGA devices, the only way to confirm a solid connection on each pad, is to perform an X-ray analysis on the device. A report by NASA [60] documents the proper procedure to use BGAs in a space environment. One of the many factors that make BGAs so hard to mount, is the lifetime of the joints between the device and the PCB. Due to expanding and contraction of the PCB due to extreme temperature fluctuations, the joints are subjected high levels of stress. These stresses eventually lead to a failure on either the device or the PCB.

Since this PCB is only a prototype, a QFP package will be used. This will facilitate easier soldering and debugging.

To accommodate both designs, the Microsemi ProAsic3E 1500 (A3PE1500) [61] device was selected, since it also offers a QFP package. It offers a total of 38 400 logic units, which is sufficient for the OpenRisc soft-core processor and extra interfacing logic. Since multiple designs will be programmed to the FPGA, the re-programmability of this FPGA is a necessity. It offers both projects a familiar developing environment, as it is the same as the Actel Fusion development kit's environment. The same FPGA is also used in one of the OpenRisc development kits and sample designs could be downloaded from the OpenRisc website and adapted for this hardware setup. The FPGA is also based on flash technology, which is more resistant to upsets than SRAM based technology. These devices are available from Asic Design Services, a local supplier, with whom this research group has a good partnership.

3.3.1.2 DUT Assisting Hardware

For either designs to operate effectively, several external hardware components are required.

For the CAN controller to be able to communicate with other CAN devices, a transceiver is required. This changes the signals from transistor-transistor logic (TTL) to a differential voltage signal. The voltage level of this signal is determined by the highest voltage device on the bus and can rise to 42V in both directions. For this design, the SN65HVD233 CAN Transceiver [62] from Texas Instruments was chosen.

Since the FPGA size is more than double that of a full TMR CAN controller, it is desirable to run two CAN controllers in the same design, allowing more SEEs to manifest. Since more than one CAN controller will be used, it was opted to use two transceivers.

In order for the OpenRisc soft-core processor to function, it required both SDRAM and SPI Flash components. Due to the popularity of the OpenRisc project, several hardware platforms have already been developed as development kits for the project. Co-incidentally the same FPGA was used in one of these boards. This development kit uses an AMIC A43L4616AV-7F SDRAM IC [63], along with the Micron M25P10-AVMN6P [64] SPI Flash IC. Since only the FPGA is going to be radiated, the use of components without space heritage was acceptable.

3.3.1.3 Controller

At the Electronic Systems Laboratory (ESL), access to the Actel Fusion Embedded development kit is possible. This kit consists of a Fusion FPGA, external SRAM and other peripherals.

During the course of this project, the web-server example project was used to test the CAN controller. The FPGA was Cortex-M1 enabled and it was used as the soft-core processor in the design. The Cortex-M1 served as a web-server, transmitting data over an Ethernet port. It sampled data from the analogue block of the FPGA, and delivered it in the form of a web-page. See Figure 3.3 for a block diagram showing the IP cores used to implement the web-server example in the Fusion development kit.

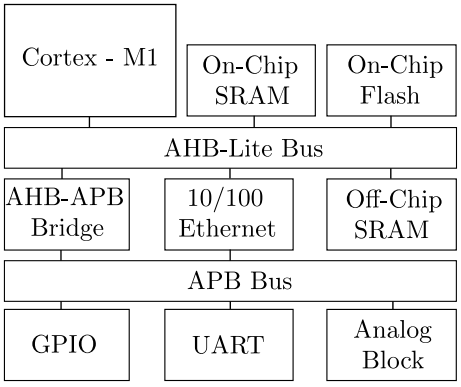

Figure 3.3 – Block Diagram Showing the IP Cores of Web-Server Implementation in the Fusion Development Kit

To save time on the designing of hardware, it was opted to use the Cortex-M1 processor as the interface controller between the computer and the DUT. Several Cortex-M1 enabled FPGAs can be found in Microsemi's catalogue. Unfortunately, the local supplier has a minimum order quantity of 20 units on all FPGAs. For this reason, it was decided to use the same FPGA as for the DUT, and both will be Cortex-M1 enabled. The Cortex-M1 will not be used on the DUT since no processing power is required.

3.3.1.4 Controller Assisting Hardware

Since the ProAsic3E FPGA does not have an analogue block or internal flash memory, external memory and ADCs are required for application memory storage and the sampling of voltages and currents on the PCB.

The support software for the FPGA offers a flexible interface between the Cortex-M1 and its external memory. This interface consists of 32 data bits and 19 addressing bits. Most SRAM and flash chips only have 8 or 16 bit data configurations. To accomplish the required 32 bits, 2 ICs were used in parallel. The IDT IDT71V416L10PHGI [65] was used for the SRAM and Micron M29W160EB70N6F [66] for the flash memory, since these were immediately available from a local supplier.

While serial communication would suffice for basic data collection, a higher transfer rate would be more desirable. One of the useful features of the Fusion development kit, is the 10/100 Ethernet interface. The web-server example also uses this interface to communicate with a computer. The schematics of the required components to implement the Ethernet interface can be found on the freely available schematics of the Fusion development kit. The implementation of the Ethernet interface requires an Ethernet Physical Layer (PHY) IC, which is available from several suppliers. For this design, the DP83848 [67] from Texas Instruments was chosen.

3.3.1.5 General hardware accessories

During testing, the health of the hardware should be monitored to ensure the correct operation of all components on the PCB. It would therefore be beneficial to have both current and voltage monitoring available on the board. This data would be fed into the controller, which will then report back to the computer for data logging. The current will be measured using an INA169 Current Shunt Monitor [68], which will be sampled by a MAX1038 ADC [69].

Damage to the FPGA due to radiation, can show up in several ways. Latch-ups and physical damage to the FPGA will result in higher power consumption. While it might be noticeable on the current monitor, the device could also heat up. It would therefore also be useful to have temperature monitoring on the board. The temperature will be sampled using a AT30TS75 Digital Temperature Sensor [70]. Two of these sensors are to be placed on the board: one below the power supply and another between the two FPGAs.

If there is an indication of a fault in the FPGA itself, it would be beneficial to be able to power cycle the DUT, without having to enter the radiation test chamber. Therefore, separate power supplies will be used for both the controller and the DUT. The controller will then be able to switch off the supply to the DUT using voltage switches in the power supply.

3.3.1.6 Limitations of the Layout

It should be noted that during the radiation experiments, only the device under test is radiated. To protect the other components, a keep-out zone needs to be placed around the FPGA.

The radiation beam will be perpendicular to the PCB surface and pass through both PCB and components. To obtain accurate results, only the FPGA should be in the path of the beam. Passive components, such as headers, capacitors and resistors are allowed to be radiated, since upsets are not expected in these components. A report on the effects of radiation on capacitors and resistors showed that capacitors will have a drop of about 10% in its capacitance after 345 kGy, while resistors did not show any change below 1 MHz [71].

A two-layer limitation was also placed on the PCB to save on production cost. There should also be several test points for debugging with big ground pads for the probes to connect to.

3.3.1.7 Component Selection for Vacuum Environment

Not all components on a PCB function correctly in vacuum. These include, but are not limited to, electrolytic capacitors and light emitting diodes (LED).

Electrolytic capacitors consist of two layers which are isolated from one another, and then rolled up to form a large capacitive layer. For aluminium capacitors, the aluminium foil is coated in an insulating oxide layer, which forms the capacitor. If a small air pocket exists in the rolled up foil, the difference in air pressure could lead to the expansion of the size of the air pocket. In other designs, a liquid is used to create a better capacitance to volume ratio.

When placing these in a vacuum, the liquid can start to boil. In both cases, a decrease in capacitance can be expected, or even a total physical or functional failure.

To avoid these risks, all capacitors should be either ceramic or tantalum capacitors. Normal small value capacitors are made from several layers of metal which are isolated with a ceramic material to form the capacitor. Tantalum capacitors consist of a small ball of tantalum, which is covered in an oxide layer to isolate it from the surrounding conducting material. Tantalum capacitors offer higher capacitance than ceramic capacitors, but are also more expensive and larger in size.

The bodies of LEDs are made out of an epoxy to form the lens. Impurities and minute air bubbles in the epoxy can pose a threat in a vacuum environment. Similar to electrolytic capacitors, the difference in pressure can cause the epoxy to form cracks due to the expansion of these bubbles which can lead to device failure as a result of movement close to the substrate.

The LEDs are a luxury during debugging and do not add a functional purpose to the PCB. The fact that the content of the chamber is barely visible through a web-cam means that the LEDs will not be of any assistance during testing. To avoid the problem of failure of the LEDs damaging other circuits, the LEDs will be placed on removable PCBs which can be removed prior to radiation testing.

3.3.1.8 Testing Chamber

The tank in which testing will be done consists of a vacuum chamber, two mechanical arms and several holes for custom connectors.

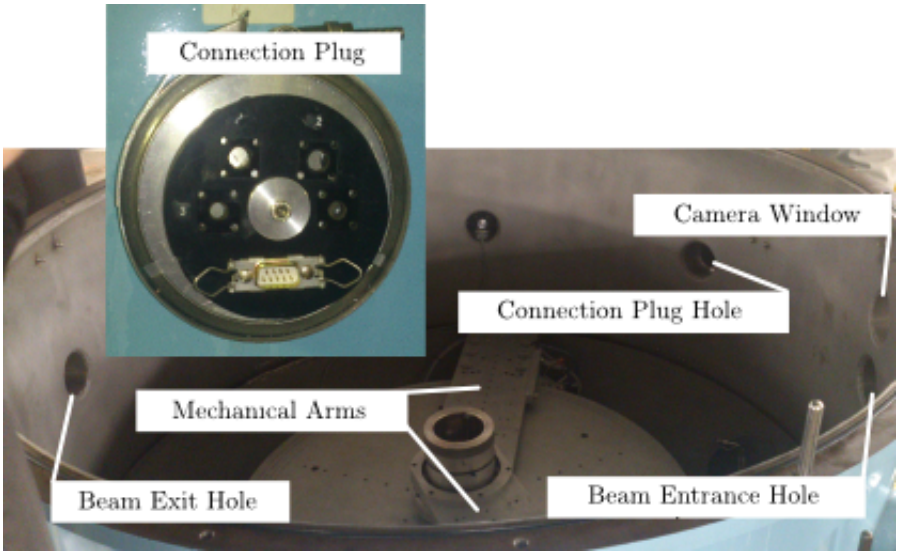

Figure 3.4 – Image Showing Detail of a Vacuum Connector and the Inside of the Scattering Chamber

To obtain optimum results, the radiation beam will be defocused, as to cover the entire die of the FPGA. While the exact size of the die is not known (due to intellectual property by Microsemi), the beam will be defocused to cover most of the chip, while not touching the other components. The hardware will be placed on one of the two mechanical arms, as to move it in and out of the beam. The other arm will be used to hold a calibration disc to help with the defocussing of the beam. As a safety precaution, all cables to and from the board, will be long enough to reach from the hardware, to the centre of the tank, and then to the sidewall. This is to avoid damage from the movement of the arms.

The sidewall has holes with a diameter of 95mm to house a custom connection plug to connect internal to external cables. This plug should be populated with vacuum rated panel mount connectors. From the hardware design, we need several connections. An Ethernet to USB hub will be placed inside the tank, providing 4 USB connections. Two will be used for the FlashPro programmers, one for serial communication with the DUT and one for a moving platform. Table 3.2 lists the proposed connectors to be used.

Connection	Proposed Connector	Description
Ethernet	DB9	Communication with hardware
Power	Coaxial	5V supply for the hardware
Ethernet	DB9	Communication with Ethernet to USB hub
CAN	DB9	Monitoring of CAN bus

Table 3.2 – List of Connections to the Hardware

To monitor the hardware visually, an Ethernet web-cam will be placed above the beam entry point into the tank. Looking through a clear Perspex window, any damage to the hardware or cables can be observed remotely via the web-cam.

A power supply will be placed outside the tank to provide the required 5V for the hardware. The power supply and Ethernet to USB hub will be shielded using a brass block.

For safety purposes the control room is located far from the test chamber. The only connections available between the two are an Ethernet cable, a 220V AC power line and several coaxial connections. Since there are more Ethernet devices than cables, a network switch will also be required next to the tank.

3.3.1.9 Control Room

Due to the distance between testing chamber and the control room, the stability of the Ethernet connection could be questionable. To overcome this, a second switch will be located in the test chamber to re-condition the signals.

The control room plays host to all the required components to manipulate the radiation beam, vacuum and arm movements inside the tank. Besides the already in position components, two additional computing stations will also be required. One of these will be used to manipulate the moving platform on which the hardware is mounted and to monitor the web-cam, the other will be used to interface with the DUT.

3.3.2 Summary of Requirements

From the previous sections, a complete list of the hardware requirements can now be compiled and is shown in Table 3.3. Combining these requirements, along with the controlling FPGA and DUT FPGA designs, a system diagram can be constructed and is shown in Figure 3.5.

PCB	**Rationale**
Keep out area	Only DUT to be radiated
2-layer PCB	Cost saving
Test points	Probing of trace signals
PCB - controller	
2x SRAM ICs	32-bit wide program memory for Cortex-M1
2x FLASH ICs	32-bit wide code memory for Cortex-M1
Ethernet PHY IC	Communication between Cortex-M1 and computer
I2C ADC	Monitoring of current and voltages
I2C Temperature Sensors	Monitoring of PCB temperatures
Voltage switches	Power cycle of DUT
PCB - DUT	
SDRAM IC	16-bit wide program memory for OpenRisc
SPI FLASH IC	8-bit wide code memory for OpenRisc
2x CAN transceiver ICs	Communication between CAN controller and CAN Bus
GPIO header	Testing of OpenRisc
SPI header	External Programming of SPI Flash
Separate power supply	Power Cycle of DUT by controller
Test Chamber	
Power supply	To supply power to testing hardware
Ethernet switch	Switching of Ethernet signals
Ethernet web-cam	Remote visual monitoring of hardware
Ethernet to USB hub	Remotely control USB devices
2x FlashPro Programmers	Programming of FPGAs
Control Room	
Ethernet switch	Reconditioning of Ethernet signals
2x laptops	Control of testing hardware

Table 3.3 – List of Hardware Requirements

3.3.3 Top Level design

Figure 3.6 shows a diagram of the test environment. Based on the design above and the hardware requirements listed in Table 3.3, the PCB was designed. Figures B.1 to B.10 in Appendix B show schematics of the experimental PCB, while Figure B.13 shows schematics of the quad channel UART daughter-board.

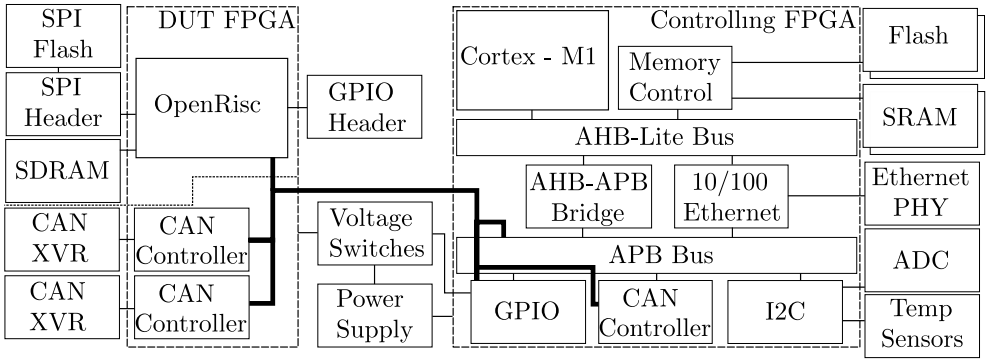

Figure 3.5 – Diagram of the Top Level Design of the Experimental PCB

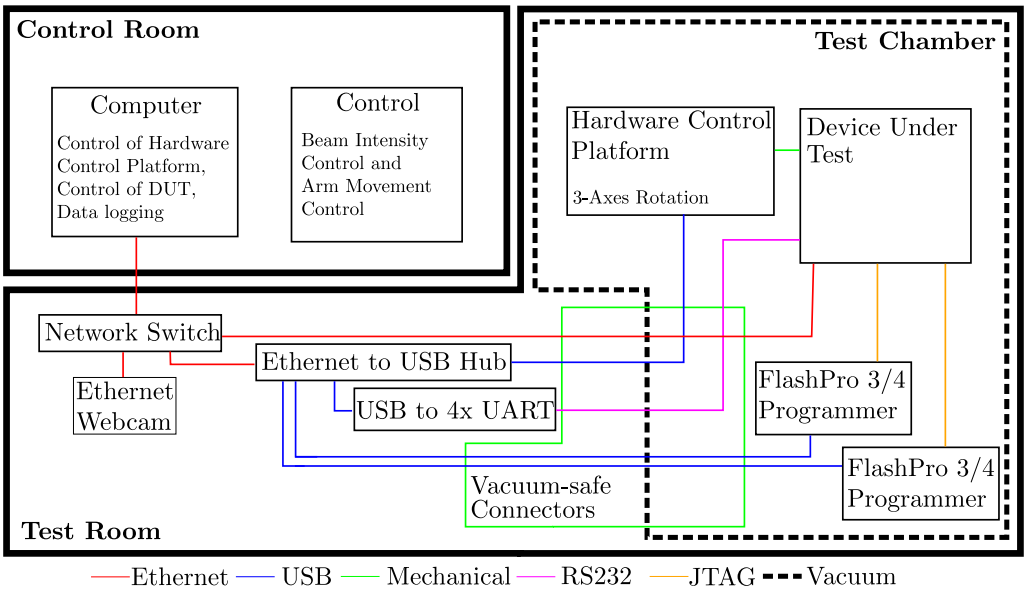

Figure 3.6 – Diagram of the Complete Test Environment

3.4 Modifications to Original CAN controller

The original Verilog implementation of the CAN controller needed certain adjustments to make it usable for this project. This required a change in the interface, as well as a re-arrangement to make it more editable.

3.4.1 Interface

The original version of the CAN controller, as found on OpenCores.org, is easily customizable using parameters in one of the source files. One of these options is to select which interface the user wants to use to interface with the CAN controller. The available interfaces are limited to the Wishbone Interconnect and the 8051 interface. To ensure a stable design, the design was based on a Cortex-M1 soft-core processor. Unfortunately, the Cortex-M1 does not have access to either of these interfaces.

It is possible to use the GPIO IP Core and bit-bang to the CAN controller, but this would involve extensive coding and would have added a big strain on Cortex M1. For this reason, an AMBA interface was added to the design. Table 3.4 lists the connections used by both the AMBA interface and the external interface of the CAN controller.

AMBA	CAN controller	Description
PADDR	addr	Addressing signals
PWRITE	we	Write not Read signal
PSEL	cs	Device Select signal
PENABLE		Device Enable signal
PWDATA	data_in	Master to Slave data
PRDATA	data_out	Slave to Master data

Table 3.4 – List of Connections for AMBA Master to CAN controller

The implementation of the APB bus has a data input signal for every sub-module. This allows the CAN controller to output data constantly on its data output pin. The CAN controller only reads data from its input pins on the rising clock edge, if both the Write Not Read (WE) and Device Select (CS) signals are high. The APB bus pulls both CS and WE high one clock cycle before pulling the Enable pin high. The missing Enable pin on the CAN controller is important to the design, as without it, the CAN controller will clock in the same data twice, causing the CAN controller to perform certain tasks twice, such as releasing the received data buffer. To avoid this problem, the PENABLE pin can simply replace the PSEL signal on the CS pin on the CAN controller.

3.4.2 Flattening of the CAN controller Tree

Two problems became evident when studying the CAN controller hierarchy. If the BSP or registers module is protected using TMR, all of the lower modules will also be triplicated. This is problematic as any protection on the lowest module will also be triplicated. If both the FIFO buffer and the BSP are protected using TMR, the FIFO buffer will in effect be replicated 9 times.

Similarly, all of the configuration registers as well as the transmit buffer are implemented using three single generic files, *can_register_asyn_syn*, *can_register_asyn*, *can_register*. Editing one of the registers would result in all the other registers using the same file to be altered.

It was for these two reasons that the design of the CAN controller was flattened out so that all instances of all files are instantiated in the top level of the design.

3.5 Mitigation Schemes

To make a CAN controller, utilizing mitigation by design, more viable for implementation in an FPGA, the area usage of the circuit should be as low as possible. Using full TMR uses three times the area of the original design excluding the voting logic required. For

this reason, several techniques will be combined to lower the footprint of the device, yet maintaining a high resistance to upsets.

3.5.1 SET Filter

The entire idea behind the SET filter is to avoid Single Event Transients latching into a memory element and becoming a Single Event Upset. Therefore, the ideal place for the SET filter is prior to a memory element. The filter should not only be placed before the data input, but also in front of all the enable signals of the memory element.

To determine where the SET filters should be placed, the net-list should be analysed to determine the number of gates prior to the input pins, as well as the longest route to a previous memory element or FPGA input pin.

The longest route will assist to give an idea of the impact the specific SET filter will have on the maximum attainable clock frequency of the design. If one inserts the SET filter on the longest paths, it may slow down the design.

The number of gates prior to a specific memory element will show the likelihood of an upset in this region. The more gates there are the more probable it will be that an upset will occur. If the SET filter is inserted in front of a pin with few elements prior to it, it actually increases the overall likelihood of a SET occurring. If there is a fifty-fifty relationship between the SET filter gate count and the number of gates prior to the input pin, there is a 50 percent chance of the SET actually occurring in the filter itself. While the SET filter can filter SETs in previous elements, it is not able to fully filter out SETs inside itself.

3.5.2 TMR

Triple modular redundancy will be implemented on two different levels in this design.

Firstly, at gate level. For this technique, TMR will be applied to the memory elements and sequential logic. This will mostly be applied to memory elements as an upset will most likely remain in the system, if it occurred in a memory element.

Figure 3.7 shows a simple triple module redundancy D Flip Flop (DFF). In the figure, all inputs are connected to matching ports of the DFF. Outputs are connected to a majority voter. The majority voting can be done in Microsemi FPGAs using a single versatile using the MAJ3 macro.

To detect when an error has occurred, the three outputs can be combined using an XOR gate, which will only output a logic '1' when the inputs are not the same value. To recover from an error, the error signal is combined with the input enable pin. A logic '1' on either of these pins will force the DFF to clock in data. The write enable pin is used as a data select signal. If the write enable pin is low, data will be fed back from the output of the majority voter. If the write enable pin is high, new data is about to be clocked in. This will enable recovery, unless new data is to be clocked in. Figure 3.8 shows this implementation.

To use the above method on an 8 bit register, each bit is approached as a single register. Each bit will have its own triplicated DFFs, majority voters and data MUX. To detect an

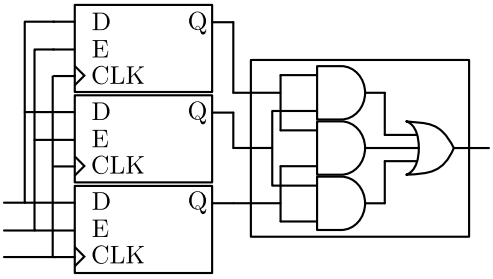

Figure 3.7 – Diagram of the Traditional TMR

error, all the error signals are combined using OR gates. If any of the bits are corrupt, all the bits will be re-written. This uses fewer gates than re-writing individual bits, since the eight bits share the same enable line.

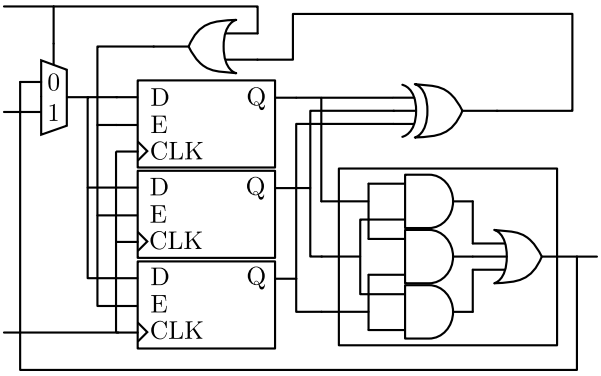

Figure 3.8 – Diagram of the Single Bit TMR With Recovery

Most of the registers being protected using TMR and recovery are not on the critical path. The addition of the MUX at the data input and the majority voter at the output should not have a noticeable effect on the maximum operating frequency.

For sub-modules being protected using TMR, only the majority voter at the output can affect the maximum operating frequency. The single majority voter should add a single ignorable delay.

3.5.3 EDAC using Hamming Code

Implementing the Hamming code's parity checks in hardware as per §2.1.3.4 can be done by using XOR-gates, as shown in Equations 3.5.1 to 3.5.4.

$$p_0 = \{d_0 \oplus \{d_1 \oplus \{d_3 \oplus \{d_4 \oplus d_6\}\}\}\} \tag{3.5.1}$$

$$p_1 = \{d_0 \oplus \{d_2 \oplus \{d_3 \oplus \{d_5 \oplus d_6\}\}\}\} \tag{3.5.2}$$

$$p_2 = \{d_1 \oplus \{d_2 \oplus \{d_3 \oplus d_7\}\}\} \tag{3.5.3}$$

$$p_3 = \{d_4 \oplus \{d_5 \oplus \{d_6 \oplus d_7\}\}\} \tag{3.5.4}$$

For an eight bit data byte, a total of 14 XOR-gates are required to implement the Hamming code encoding, as shown in Figure 3.9. The Hamming Code word is produced by routing the data bits and parity bits to the correct position.

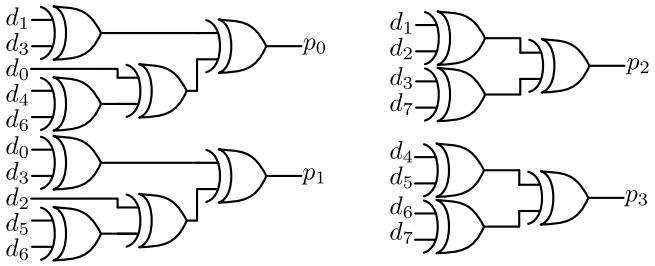

Figure 3.9 – Diagram of the Hamming Code Encoder's Parity Calculator

In order for the decoder to calculate which bit was upset, the parity bits first need to be recalculated. Comparing the new parity bits to the stored ones will give the corresponding position of the faulty bit, as shown in Figure 3.10. In the event that a data bit is flipped, the flip can be repaired using an XOR-gate. Using AND gates with inverted inputs, the exact combination of position bits will result in the faulty bit being flipped, as shown in Figure 3.11.

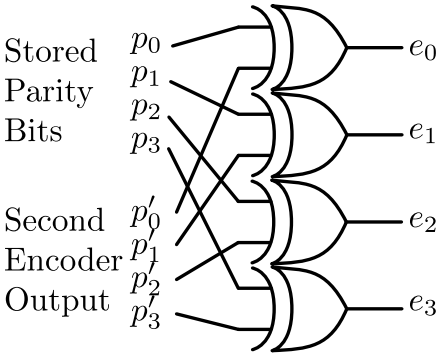

Figure 3.10 – Diagram of the Hamming Code Decoder's Wrong Bit Identification

To implement the decoder for an eight bit data type, a second encoder is required to recalculate the parity bits, along with 28 more gates. This results in a total of 42 gates for the decoder.

3.5.4 Mitigation Scheme Design

Following the analysis of the CAN controller in §3.2 and weighing the effects of different mitigation techniques on the original circuit, a combination of mitigation schemes were applied to different parts of the CAN controller.

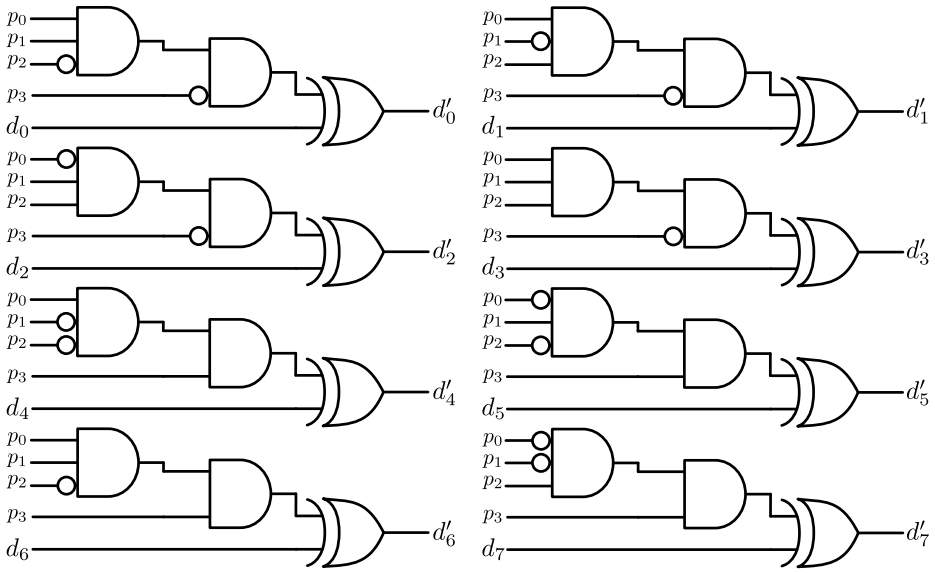

Figure 3.11 – Diagram of the Hamming Code Decoder's Bit Flip Mechanism

3.5.4.1 CAN TOP

The *can_top* file mainly consists of the instantiation of all the sub-modules, but also declares the connections to the controller. For this project, the CAN controller is not located on the same FPGA as the processor. The traces, I/O pins and I/O buffers are all susceptible to upsets. All I/Os to the CAN controller will therefore be protected with SET filters.

3.5.4.2 CAN BSP

The *can_bsp* file is the core of the CAN controller. This file contains the state machine for sampling the data and error generation. Due to the high importance of the state machine, this BSP module will be protected using TMR. While a full TMR of the BSP module is sufficient, it is possible for all three state machines to break. The possibility of interconnecting the state machine should also be investigated.

3.5.4.3 CAN FIFO

The *can_fifo* file consists of three FIFO buffers, each 64 bytes in length. Due to their large size, neither DMR nor TMR is appropriate. The *can_fifo* file makes up roughly 40% of the CAN controller. Data inside the FIFO buffer will remain constant for long periods of time and are not crucial to the functionality of the CAN controller. While the controller should not let the data sit in the FIFO buffer for a long time, it should be kept safe. Using Hamming code, the data bytes and information nibbles will be protected against single bit upsets. Should an upset occur between the time a message is received, and the FIFO is serviced, a single bit per byte can still be repaired. The repair occurs during the read clock cycle and since there is no other sequential logic between the register output and the output MUX, the delay added should be minimal. The data buffer will be protected using Hamming

code {12, 8}, while the information buffer will be protected using Hamming code {9, 5}.

3.5.4.4 CAN CRC

The *can_crc* file consists of a CRC encoding module, which is used to check whether the data received is valid. As it only consists of 52 logic cells, while having a important function, it will be protected using TMR.

3.5.4.5 CAN ACF

The *can_acf* file consists of several combination logic cells, converging into a single output. It is possible to protect this section with a single SET Filter.

3.5.4.6 CAN BTL

The *can_btl* file consists of all the logic required for synchronizing and sampling the CAN bus. It amounts to 192 logic cells, but is one of the most important components of the design and needs superior protection. Since the BTL is a time critical circuit, it will be protected using TMR to avoid any timing issues.

3.5.4.7 CAN Registers

The *can_registers* component consists of mainly long-term configuration registers, with a few short term registers. Most of the short term registers pertain to the generation of an interrupt. Reading the interrupt register will release the interrupt. In the event of an upset in the interrupt register, investigation by the processor should show that it was incorrect and that it can be ignored. While a misplaced interrupt will not interfere with the functionality of the device, it will cause some overhead in the processor, as the processor needs to check the source of the interrupt.

The long term registers should remain fixed at all times during operation. A single upset to the bus timing will result in the CAN controller going out of sync. Once out of sync, it will miss all the messages and another device could hold the CAN bus in a busy state. Therefore, the long term registers should be protected.

The following registers will be protected using full TMR with recovery:

Acceptance Code 0 to 3 - Important for message accepting
Acceptance Mask 0 to 3 - Important for message accepting
Bus timing 0 and 1 - Important for BTL
Clock divider bit 7 - Basic/Extended mode selection
Error warning - Exceeding error count can put device in reset
Interrupt enable - Missed interrupts can flood device
Mode - Reset device etc.

The following registers will not be protected:

Clock divider lower bits - The external clock generation is not important in the FPGA environment. For the SJA1000 IC, the external clock can be used to drive slower peripherals. Scaling down the FPGA clock can easily be done using the internal Phase-Locked-Loops (PLLs) of the FPGA.

Command - Since it will be cleared within the following few clock cycles

Transmission data - Since the controller should not store the data for a long time before transmitting

Full TMR and recovery as in §3.5.2 will allow the protected registers to be reconfigured if an error is detected.

3.6 Final Implementation

Applying the mitigation schemes to the CAN controller will undoubtedly increase the footprint of the design. To make the solution viable, the area usage on the PCB needs to be as small as possible to replace a separate CAN controller with an FPGA.

3.6.1 Logic Cells Required

It is not only important that the CAN controller fits on the FPGA, but some control logic should also be added. An estimation of the area usage of the CAN controller is roughly 200% of the original, the usage is expected to be roughly 11 000 logic cells. To communicate with the CAN controller using a microprocessor, an interface needs to be implemented. If the CAN controller is used to connect a sensor to the CAN bus, a state machine needs to be implemented which can initialise and control the CAN controller and interface with the sensor. A rough target for the CAN controller and control logic would be around 15 000 logic cells.

3.6.2 Possible FPGAs

Possible FPGAs to consider when implementing the mitigated CAN controller in a final satellite system design include the Actel Igloo/e, Igloo 2 and ProASIC3 range of FPGAs. All of these FPGAs are flash-based and include the basic required physical attributes to implement the CAN controller.

In the Igloo/e range of FPGAs, the smallest footprint possible for an FPGA with enough logic cells is the FG144. This footprint is a BGA of 12 x 12 pins requiring a total of 13 mm x 13 mm of area on the board. The M1/AGL600 has 13 824 logic cells, which will fit a minimalistic design. The M1/AGL1000 has 24 576 logic cells, which will accommodate a CAN controller and controlling logic comfortably.

In the new Igloo 2 range of FPGAs, the smallest footprint is the VF400. This is a pinned packaged requiring 17 mm x 17 mm of PCB area. The M2GL025 has 27 696 logic cells. Microsemi is currently developing a smaller form factor package and they are aiming at sub 14 mm x 14 mm PCB footprints.

Other possible devices include the ProASIC3 A3P1000 or A3P1000L. Both these devices offer a total of 24 576 logic cells, on a FG144 footprint. The ProASIC range of FPGAs do however use more power, but can attain a much higher operating frequency.

While the maximum frequency for the original CAN controller is roughly 55 MHz in a ProASIC3E A3PE1500 FPGA, the mitigated design will undoubtedly be slower. Implementing the same design in the Igloo AGL1000 FPGA results in the maximum frequency dropping to roughly 38 MHz. In this design the clock frequency used will be 20 MHz. If the Igloo is to be used, the maximum frequency of the CAN controller after mitigation could fall below the clock input. This will result in unpredictable behaviour.

3.7 Testing the Design

There are various ways to test for SEUs. This section covers the design of the tests as to detect SEUs in as many areas of the CAN controller as possible.

3.7.1 Radiation Testing

Ideally, all the testing will be done inside a radiation chamber using the designed hardware. Two boards will be manufactured: one for this project and another for Francois' project. Although two boards are available, the time it takes to replace the board in the vacuum tank is too long and is therefore the last resort. After a round of test for this project, Francois will reprogram the board to suit his tests.

Each project will have it's own individually developed testing software. The software for this project will be written in C using QT Creator. This application will utilize the Ethernet connection to control the soft-core microprocessor and query the CAN controllers. The following sections will describe the two test designs proposed.

3.7.1.1 Configuration Hold Test

In this test, all writable registers will be given a known bit pattern. All readable registers will be read periodically, storing the values for both future analysis and comparison to the next read value. In the event of sequential values having different values, an error has occurred.

3.7.1.2 Operation Test

In this test, all CAN controllers will be set to extended mode and will conduct normal communication. All devices will send and receive messages of random lengths containing random data, while monitoring all configuration registers and error registers for abnormalities, which will indicate radiation induced errors.

3.7.2 Simulated Testing

At a late stage in the project, beam scheduling changes and test set-up problems occurred resulting in radiation testing being delayed to close to the submission deadline. As a precaution, it was decided to focus more on simulated testing, specifically including detailed error injection simulations.

3.7.2.1 Injecting SEEs

The simplest way to inject SETs into the design is using the simulation test-bench. Using the design hierarchy, sub-modules' wires and registers can be accessed and manipulated inside the test-bench. Out of previous experience, this method of injecting only works during pre-synthesis simulation. All simulation needs to be done at least post synthesis, but ideally post layout.

When running a design pre-synthesis, no timing information is available. The Verilog is translated into a functional model, and all simulated signals are based on this model. With post synthesis, the Verilog is translated into physical elements. The delay for each element is known and can therefore be used in the simulation. This allows the simulation to show whether or not the design will have timing issues, depending on the maximum frequency at which the design can be run at. Post layout takes this one step further. After layout, the software calculates the length of each trace used inside the FPGA, and adds the delays accordingly. It also adds delays for routing elements. When all this information is available, the simulations are an accurate representation of what should be happening in the FPGA after programming. The only factor that cannot be included is the die quality. The way the software compensates for this, is to have the user select which grade device he wants to use. For the selectable grades, the worst case timing delays are used in the simulation.

The other method of injecting faults is using the *run.do* file generated for the simulation. It is essentially a list of commands which is sent to the simulating software. Using the commands listed in Listing 3.1 prior to it calling *run -all*, will force the simulation into performing tasks which are not possible through the test-bench.

```
1 force [-cancel [@]<time\_info>]
  <signal\_name> <signal\_value> <time\_info>

  add wave <signal\_name>
```

Listing 3.1 – Possible commands for *run.do*

Using *force*, the value of any register or wire can be forcefully changed at a specific time. The forced value remains until the *noforce* command is performed. If a *-cancel* parameter is defined, the force command will expire after the given time [72]. Using *add wave*, only the signals required by the user can be added to the simulation window [72].

3.7.2.2 EDAC on FIFO

To verify the working of the EDAC encoders on both buffers, several SEUs need to be injected to simulate different events. For this test, bit flips in both directions will be performed on both the data and Hamming code bits. Starting with one bit at a time, and eventually ending with two bits being flipped.

Theoretically, independent of the bit changed and the direction of the flip, the data output from the FIFO should remain correct. In the event of two bits being changed, the output should be incorrect. This is the result of the Single Bit Correction of the Hamming code. For this test, only the data FIFO buffer was tested. Using the AMBA interface, data can be read from the FIFO. The information FIFO cannot be read as easily, as it is only used when releasing data from the data FIFO.

3.7.2.3 TMR on Registers

Several of the registers were protected using a full TMR and recovery block. Testing the implementation on one of these registers should confirm the functionality of the TMR and recovery, as well as the implementation thereof.

To verify the functionality, the value of one of the three bits needs to be changed. The majority voter would then emit the correct value, while the recovery logic detects a fault. This fault is then combined with the write enable signal to route data from the output of the majority voter, back to the input of the register. The faulty bit will then be overwritten on the next rising edge of the clock.

Writing to the register will write to all three bits. Theoretically, the value of the register should not change. In the event of two faults in one clock cycle, the value should change as the majority voter will vote on the two faulty values. However, simultaneous faults in two different logic cells are highly unlikely.

Chapter 4

Implementation

In this chapter, the detailed implementation of the hardware design will be discussed. This involves the hardware layout, population and testing of components, as well as the development required due to changes in the design. Once the hardware is completed, the FPGA design and improvement, implementation and analysis of the mitigation schemes will be discussed, followed by the implementation of the mitigation schemes inside the CAN controller. This chapter will finish off by discussing the testing of the CAN controller and mitigation schemes, both in simulation and radiation testing, using the developed simulations and testing software.

4.1 PCB design

This section describes the design of the PCB as shown in Figure 4.1. Subsections of this image will be discussed.

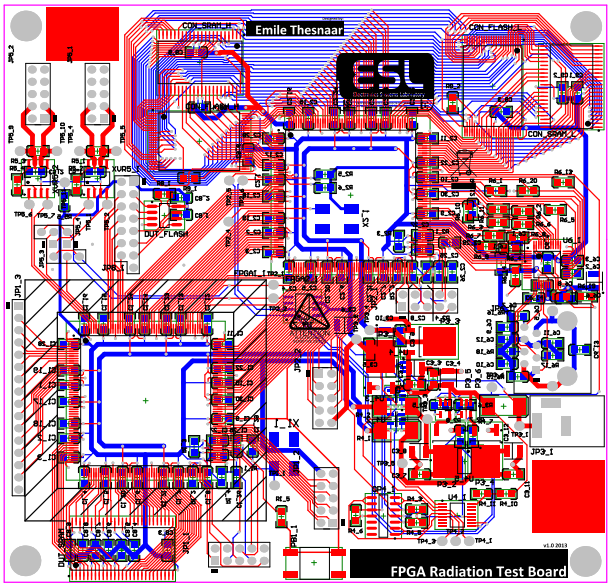

Figure 4.1 – Diagram of the Experimental PCB Design

4.1.1 Device Under Test

As per Table 3.3, the DUT is required to have SDRAM and SPI Flash for Francois' OpenRisc processor. Figure 4.2 shows the SDRAM chip at the bottom and the SPI Flash at the top. The GPIO header, which can be used to interface with the DUT FPGA, can be seen on the left. The DUT FPGA and GPIO's schematics can be found in Figure B.2 and the SDRAM's schematics in Figure B.9 in Appendix B.

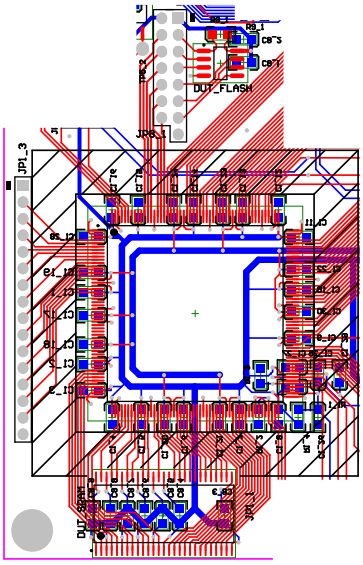

Figure 4.2 – DUT With SPI Flash, SDRAM and GPIO Header

The header to the left of the SPI Flash disconnects the signals to the SPI Flash. For normal operation, jumpers will be set to connect the pins to one another. For initial debugging an external SPI programmer can be connected to the header. The SDRAM was incorrectly labelled DUT_SRAM in both the schematics and on the PCB.

Since this is a high speed design, all traces were spaced with the same amount of spacing to minimize crosstalk. All de-coupling capacitors are placed as close to the pins as possible to reduce noise on the power supply from interfering with the operation of the IC. Since the capacitors are passive components, they are allowed inside the keep-out zone.

To prevent the size of the PCB from becoming too large, the SDRAM IC was moved slightly into the keep-out zone. The keep-out zone is not an exact indication of where the beam will radiate. It is therefore acceptable for pins of the SDRAM to overlap with the keep-out zone.

The DUT also requires two CAN transceivers according to the requirements listed in Table 3.3. Figure 4.3 shows the CAN transceivers above the FPGA, with the schematics listed in Figure B.6.

Since the CAN signals are differential signals, the traces are symmetrical around the centre of the transceiver and the header. Although these traces are quite short, it is good practice to keep differential signals the same length to avoid possible timing issues.

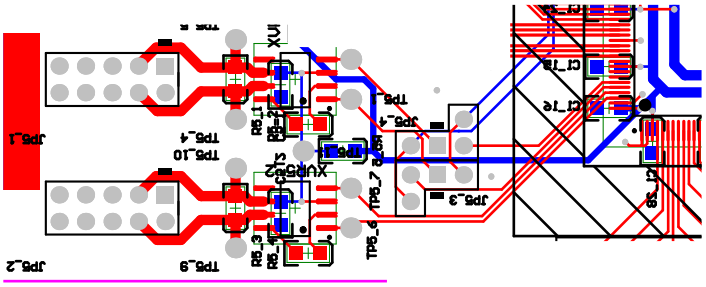

Figure 4.3 – DUT With CAN Transceivers

Between the FPGA and transceivers a header can be seen which selects with which FPGA the second CAN transceiver communicates. This is to allow the controlling FPGA to interact with the CAN bus, using an unmitigated CAN controller. This will be useful to send and receive CAN messages from a reliable source. In theory, the CAN controller on the controlling FPGA will not be influenced by the radiation. The jumpers can also be configured to connect a CAN controller on the DUT to one on the controlling FPGA.

4.1.2 Controller

In Table 3.3, the controller requires two SRAM and flash chips to run the Cortex-M1 processor. These need to use the same address bus, but the data bus will be divided into upper and lower 16 bits. The schematic for the controlling FPGA is shown in Figure B.3, while the memory is shown in Figure B.10 in Appendix B.

The lower 16 bits' flash and SRAM ICs are located to the right of the FPGA, while the upper 16 bits' ICs are located to the left in Figure 4.4.

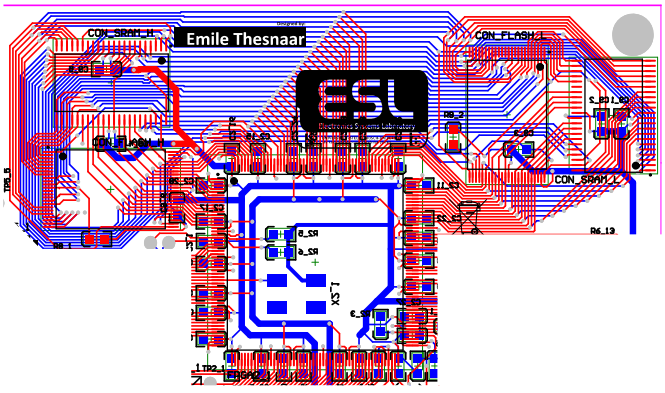

Figure 4.4 – Controller With 2x SRAM and 2x Flash

Similar to the memory of the DUT, all the traces where placed with the same amount of spacing where possible. This is to minimize crosstalk and to keep the inductance as low as possible and will hopefully avoid corruption of the data signals.

4.1.3 Power Supply and Monitoring

A separate Power Supply Unit (PSU) is required for the DUT to allow the user to switch off the DUT as per the requirements in Table 3.3. The schematics for the power supplies are shown in Figure B.4, with the monitoring schematics in Figure B.5 in Appendix B.

The input to the board is run through a sense resistor, which is monitored by an INA169 IC to measure the current. The output from the INA169 is first sent through a buffering op-amp, before connecting it to an I2C ADC. The input is also connected through a voltage divider to the ADC to measure the voltage. Similarly, the outputs from both regulators for the DUT are also measured, since no voltage divider was added.

The ADC was based on the 5 V version of the ADC. Once the design was completed the 3.3 V ADC's data sheet was consulted and the rail voltage was changed to 3.3 V. This means that the maximum voltage readable is slightly less than 3.3 V and that the 3.3 V line cannot be accurately measured.

4.1.4 Populating and Testing the PCB

To confirm that the design of the PCB is working properly the components were populated and tested in several phases.

4.1.4.1 Phase 1: Power

This phase consisted of populating all the components required to supply, regulate and filter power to the board. These components are populated separately from any other active components. Since the voltage regulators have the same footprints, the voltages need to be confirmed prior to populating active components. It is also required to mount the sensing resistors from the monitoring phase in order for power to be supplied to the voltage regulators.

To enable the testing of these components a multi-meter can be used to measure the voltages on the test points. It should also be noted that the voltage switches might be switched off. To force them on or off a voltage should be applied to TP3_6.

Measure Point	Value
TP3_1	Voltage from source
TP3_2	3.3V
TP3_3	3.3V (when TP3_6 is pulled high)
TP3_4	1.5V
TP3_5	1.5V (when TP3_6 is pulled high)
TP3_6	Floating

Table 4.1 – List of Measurements at the End of Phase 1

4.1.4.2 Phase 2: Controller FPGA

This phase consists of populating the controlling FPGA, oscillator and programming header. For stability of the FPGA, or any other IC for that matter, decoupling capacitors need to be added near every supply pin. Due to the large size and complexity of the ProAsic3 A3PE1500 device, it requires eighteen 3.3 V and nine 1.5 V supply pins. A single error on one of these capacitors could damage a FPGA, costing roughly R1 600.

To test whether the controlling FPGA is working correctly, a simple design was created which scales down the 100 MHz clock and toggles LEDs on the daughter-board.

4.1.4.3 Phase 3: Controller Memory

This phase consists of populating the controlling FPGA's memory, to run the Cortex M1. Once the FPGA is confirmed to be operating as required, all the extra components can be added to complete the functionality of the controller. This includes the SRAM and flash ICs.

To test whether the controlling FPGA is working correctly, a design with the Cortex-M1 was created that should run code from the flash memory. At this stage, it was discovered that the FPGA was not Cortex-M1 enabled. For this reason it was not possible to use the Cortex-M1 processor. Instead, the Core8051s processor was used as alternative. Whilst the Cortex-M1 uses the AHB bus to interface with memory, the Core8051s does not, but it has its own custom memory interface. To apply it in this hardware design, a memory interface had to be developed. See §4.2.1 for a detailed discussion on the change to the Core8051s and the custom memory interface.

With the new Core8051s programming the flash memory is more complicated. Loading the program memory into the flash memory is done by debugging the soft-core processor using SoftConsole. With the Cortex M1, CoreMemoryManagement handles the memory interface. The Core8051s processor manages its own memory. SoftConsole uses a programming Sprite, which is an Extensible Mark-up Language (XML) script, to control the soft-core processor and program the memory. Since this design is not based on the same memory as featured on the development kits, a custom programming Sprite had to be developed. See §4.2.2 for a detailed discussion on the Sprite development.

Due to this change, the test project was adapted for the Core8051s. Using CoreGPIO, a C program was written which toggles the LEDs. This verifies that the flash memory stores the program and the SRAM stores the variables during execution.

4.1.4.4 Phase 4: Monitoring

This phase consists of populating and testing the components required to monitor the voltages, currents and temperatures on the board.

As specified in Table 3.3 there will be an ADC converter to sample the voltages and currents. The power input to the PCB goes through a 0.1 Ohm resistor. If the board draws 1A of current, the voltage drop over this resistor will be 0.1V. While normal 0805 surface mount

resistors have a rating of 0.25 W, sensing resistors need a higher rating. For this reason a 5 W 2512 resistor was used.

To measure the current, an INA169 current shunt monitor was placed over the resistor. The current measurement can be calculated using Equation 4.1.1.

$$V_o = \frac{I_s \cdot R_s \cdot R_l}{1000} \tag{4.1.1}$$

The R_l resistor is located between the output and ground of the INA169. Placing a 33 kOhm resistor results in a gain of roughly 3.3 V/A. This means that for every 1 A flowing through the sensing resistor, the INA169 will output 3.3 V.

Due to the fact that all input pins to any component have an internal resistance, the output from the INA169 needs to be buffered. Since the three different currents will be measured, the buffer was implemented using a quad channel operational amplifier. Using negative feedback and connecting the output to the positive pin of the op-amp, a unit gain buffer was implemented. This INA169 along with the buffer circuit can be seen in Figure B.5 in Appendix B.

4.1.4.5 Phase 5: Ethernet Interface

Because of the change from Cortex-M1 to Core8051s processors, it was no longer possible to use Core10100Apb from the Libero library. The Core10100APB connects to both the AHB and APB buses. The AHB bus is used to interface directly with the memory, while the APB bus is used for communication between the processor and Core10100Apb.

The driver for the Core10100Apb initializes buffers in the memory of the processor. The start address and length of these buffers are then passed to the Core10100Apb through the APB bus. Whenever a message is received from the Ethernet PHY it is stored directly in memory. Similarly, when a message is transmitted, the data is read directly from memory.

Since the AHB bus is no longer present, there is no way to connect the Core10100Apb to the Core8051s' memory. This makes the use of the Ethernet impossible and although attempts were made to circumvent this problem in §4.2.3, it was not successful.

To communicate with the board a new interface had to be created. For debugging purposes a header was connected to each FPGA to allow removable LEDs to be plugged in. A separate PCB design was made that uses the FT4232H [73] quad channel USB to UART IC for communication between the computer and the PCB. This PCB uses female headers at the bottom to connect to the FPGA PCB and male headers at top to connect to the removable LEDs. This allows the stacking of the LED boards on top of the UART boards.

4.1.4.6 Phase 6: DUT

The test for the DUT is similar to the test of the controller as described in §4.1.4.2. The only addition is that the controller should toggle power to the FPGA.

During testing it was found that the LEDs did not completely switch off when the power to the DUT was disconnected. Measuring the voltages on TP3_3 and TP3_5 revealed that there was still voltage being supplied on these lines. Upon inspection, it was found that the DUT FPGA was being back-powered by the bus from the controlling FPGA.

To achieve this the bus signals were pulled low whenever the controller's DUT_POWER_DOWN pin was pulled low. A simple AND gate between the DUT_POWER_DOWN and the each signal internal to the FPGA was sufficient to fix this problem.

4.1.4.7 Phase 7: DUT Memory

To test the DUT memory, Francois used a simple UART to SPI programmer in the fabric of the DUT FPGA to write data to and from the SPI flash memory. Once the SPI flash memory was found to be working, the OpenRisc soft-core processor was programmed onto the DUT FPGA. Using code which was written to the SPI flash, the OpenRisc processor was able to execute the code and toggle the LEDs.

4.1.4.8 Phase 8: DUT Transceivers

To test the transceivers two CAN controllers were programmed onto the DUT FPGA. The controlling FPGA used the bus between the two FPGAs to interface with the CAN controller and commanded them to transmit data. The CAN bus was then monitored using a PCAN device which converts CAN to USB.

An inconsistency was found when the CAN controllers were transmitting, as the bus remained in an inactive state. Closer inspection showed that the jumpers used to route the transmission and reception lines to the transceiver were not making contact. The contacts inside the jumpers where simply bent closer to ensure a better connection and the problem was solved.

4.1.4.9 Completed Board

Once the design was completed the issue of possible overheating was investigated. To ensure that long test runs would be possible, the PCB had to stay within the operating temperature range of the components. To determine whether additional cooling was required for the power supply, the PCB was placed in a vacuum. For this test, the vacuum was created using only a rotary vacuum pump and a vacuum of $1.5 \cdot 10^{-4}$ mbar was reached. By removing the air from the vacuum chamber, convection cooling of the PCB is no longer possible. Ambient temperature at the time of test was 14^oC.

To take temperature readings the board was operated without the LED daugther-boards attached and while running the Core8051s on the main FPGA. Two CAN controllers where programmed to the DUT FPGA to send messages to one another.

From Figure 4.5 it is evident that the PCB temperature will not rise more than 16^oC degrees above ambient temperature. While the actual temperatures of the voltage regulators may

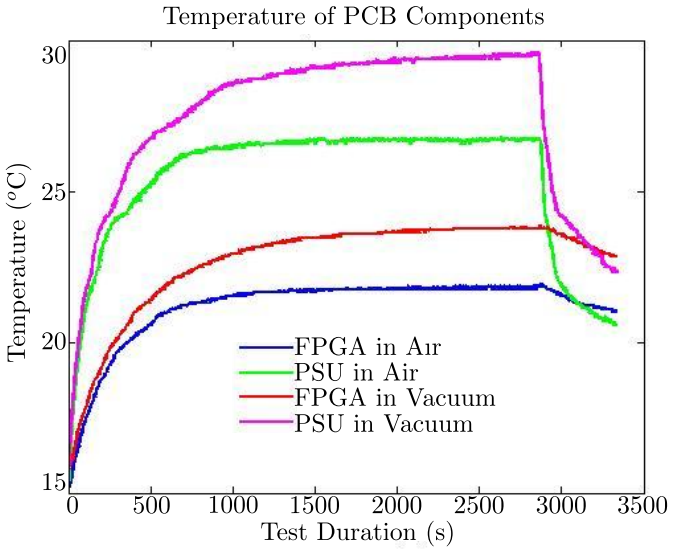

Figure 4.5 – Plot of Temperatures During Temperature Tests

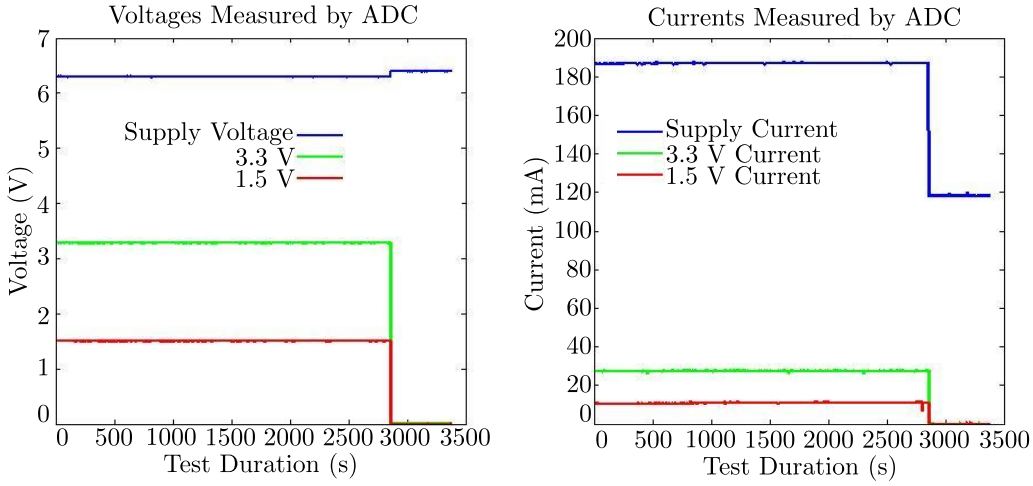

Figure 4.6 – Plot of Voltages and Currents During Temperature Tests

be slightly higher, these will not be more than 5^oC warmer than the PCB. Since the testing rooms at iThemba Labs are air conditioned to 20^oC, the maximum temperature that the PCB will reach is roughly 36^oC. This is well below the maximum operating temperature of all the components on the PCB which is 70^oC.

Using values from Figure 4.6 the power dissipation can be calculated. With an input voltage of 6.294 V, the voltage drop inside the 3.3 V and 1.5 V regulators are 2.994 V and 4.794 V respectively. With a current of 27 mA and 12 mA respectively flowing through the DUT's regulators, 154 mA is flowing through the controller's regulators. Assuming the same current flows through the 1.5 V regulator of the controller, then 142 mA flows through the 3.3 V regulator. This assumption can be made only because the FPGA core draws current from the 1.5V line. The total power dissipated is summarized in Table 4.2.

Regulator	Voltage Drop	Current	Power Dissipated
Con 3.3V	2.994V	142mA	0.425W
Con 1.5V	4.794V	12mA	0.058W
DUT 3.3V	2.994V	27mA	0.081W
DUT 1.5V	4.794V	12mA	0.058W
Total DUT On			0.622W
Total DUT Off			0.483W

Table 4.2 – Power Dissipation

4.2 Required Development Due to Changes in Design

Due to changes in the design during the implementation, further developments had to be made.

4.2.1 Custom Memory Interface

Since the Core8051s uses its own memory interface rather than CoreMemControl like the Cortex-M1, a custom memory interface had to be developed.

The Flash and SRAM used on the board use active low signals for writing and reading, while the Core8051s uses active high signals.

Because there is no easy way to determine whether the SRAM or Flash is busy, the acknowledge signals for the Core8051s had to be faked. In the configuration, the Core8051s was set up not to wait for these acknowledge signals, but to wait for a fixed number of clock cycles before reading the data from the bus.

The SRAM and flash ICs have chip select, chip enable and read-not-write signals, as opposed to the read and write signals of the Core8051s. To perform a write operation, both a chip select and a write enable signal had to be pulled low. Since the Core8051s was designed for a single Flash and SRAM chip, it did not have a chip select for either of the two chips. To create a chip select signal, the read and write signals to each chip were combined. The write and read signals also had to be clocked in after the chip select was driven low. To accomplish this, the signals were first clocked into a register and only clocked into the SRAM or flash one clock cycle later. An extra wait state also had to be added to configuration of the Core8051s.

The address and data bus widths of the Core8051s are 16 and 8 bits wide respectively. For the Cortex-M1, these were designed to be 19 and 32 bits each. The address and data buses therefore had to be trimmed and padded to avoid any undesired behaviour.

4.2.2 Programming Sprite

While writing to SRAM is a trivial task; writing to flash memory is not as simple. Because a flash cell requires a high current to program, the programming unit of the flash memory IC needs to know prior to the write operation of the intention to write to it. During this time a charge pump accumulates charge to provide the required high current burst.

Address	Data
0x0555	0xAA
0x02AA	0x55
0x0555	0xA0

Table 4.3 – Commands for Writing a Byte to the Flash device

Address	Data
0x0555	0xAA
0x02AA	0x55
0x0555	0x80
0x0555	0xAA
0x02AA	0x55
0x0555	0x10

Table 4.4 – Commands for Completely Erasing the Flash device

To start the charging process several commands need to be sent to the flash chip. For the specific flash chip used, the command structure is shown in Table 4.3 and Table 4.4. When writing to the device using Table 4.3 two address bytes and one data byte should be sent immediately after the command. While these commands are not the same on all flash devices, mostly all flash devices have this form of programming and erasing.

To program the flash memory using the debug interface on the Core8051s, SoftConsole uses a Sprite which controls the Core8051s during debugging. SoftConsole only provides Sprites for the devices which are used by Actel in the development kits. Since non-standard memory was used, none of the Sprites were usable.

The Sprite has the form of an XML document. Certain parameters and functions are defined inside. Such parameters include the device and manufacturer IDs, memory block sizes, counters, programming commands and erasing commands. Functions include programming, block & device erasing and device locking and unlocking. For simplicity, only programming and device erasing functions were implemented with only the essential functionality thereof. Refer to the Sprite's source code in Listing C.1 in Appendix C.

4.2.2.1 Erasing

To keep the Sprite simple the flash memory device will always be completely erased prior to programming. When the programming data is generated by SoftConsole, any difference to the previously programmed data will trigger a complete device erase.

To erase the device the six commands for erasing are sent. To avoid the Sprite continuing with programming while the device is still busy erasing, data is constantly being read from the flash device. During any operation on the flash device reading data will return a status byte which can be used to monitor the process, independent of the address specified. While it may be important to monitor this procedure, for this exercise it is only important to wait for the completion of the erase process. No error checking will be done. After the device is completely erased, all data fields will be set to 0xFF. To see whether the device

has finished erasing, the data is read back until the Sprite receives 0xFF. There is also a time-out attached to this process to inform the user in the event of a failure. In the event of a failure the flash device will remain in this erase process and a power cycle will reset the device to normal waiting operation.

According to the data sheet of the memory, the erase procedure should take roughly 20 seconds for a new device. As the device deteriorates over time the duration of the erase process will increase. For the current device, the device erase takes roughly 25 seconds.

4.2.2.2 Programming

When programming the device, SoftConsole sends packets of data to the Sprite to program individual bytes. Prior to programming individual bytes, the three programming commands need to be sent. When the Sprite receives a block of bytes to write, a FOR-loop is entered where a counter will count the number of bytes programmed and exit when the transfer of the block of bytes has completed.

Another way to achieve this is to unlock the device, after which it will remain in a write mode and will require one command less to program a byte. Although this will speed up the programming of the device, accomplishing this with limited knowledge of Sprites is complicated.

Because of all the extra commands being sent and the Sprite programming individual bytes, the programming procedure takes a long time. This could be shortened using the previously mentioned technique or by using a burst write mode. Once again, the complexity of the task was not worth the effort. Current programming of the devices takes roughly 50 seconds to complete.

4.2.2.3 Usage

To use the newly developed Sprite, it was placed in the *SoftConsole/Sourcery-G++/share/sprite/flash* directory. To tell the SoftConsole compiler to use this sprite, a text file is placed in the root of the project. This file contains a command telling SoftConsole to use the above created Sprite, as well as the origin and length of the data to send to the sprite. The user also needs to specify that the Memory Map Generator should use the Spritefile in the SoftConsole project when compiling the project.

4.2.3 Attempts at Fixing the Ethernet

Due to the change in processor as discussed in §4.2.1 the AHB bus was removed and using the Core10100APB was no longer possible. Two attempts were made to circumvent this problem.

The first attempt was to create a new interface for the Core8051s so that it connects to the AHB rather than directly to the memory as in Figure 4.7. The problem with this was that Core8015s has the ability to wait for the memory on the read command, it does not wait when writing data.

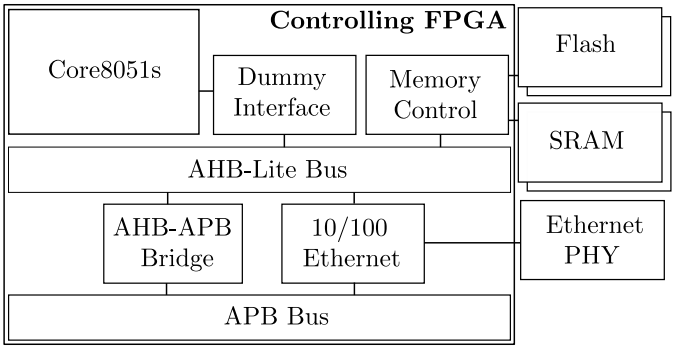

Figure 4.7 – Diagram of the Dummy Interface Between Core8051s and AHB Bus

The second attempt involved using FPGA fabric to implement a buffer to which the Core10100APB could write the data as in Figure 4.8. When an interrupt is generated the Core8051s uses the slower APB bus to then read from this buffer. Due to the complexity and fast approaching deadlines, this attempt was abandoned. Instead, a quad channel USB to UART board was developed and used to interface with the board.

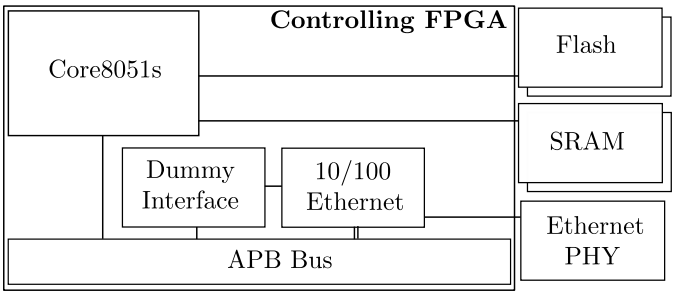

Figure 4.8 – Diagram of the Dummy Memory for Core10100

4.3 Vacuum Connector

According to the initial design, the vacuum connector required three DB9 connectors and one coaxial connector, as listed in Table 3.2. Since the Ethernet connection is no longer available on the PCB, the DB9 connection for the Ethernet was removed.

During testing, it was also found that the USB hub does not support the FlashPro programmer. When connecting a FlashPro programmer to the USB hub, the software reported that the programmer is a high bandwidth device and that it will not be able to connect to it. It was decided to place the programmers outside of tank and programming will be done inside the testing room. Two DB15 connections are required for the programming cables necessary to reconfigure the board.

Arno Barnard was kind enough to develop the vacuum connector. The connector consists of a solid aluminium body, with holes cut out for the DB9, DB15 and Coaxial connectors. To allow the vacuum connector to be used again later, it was decided that the connector

will host four DB9, three DB15 and two coaxial connectors using Joint Test Action Group (JTAG) connections.

To save cost, off-the-shelf connectors were used along with a slow curing resin to create the vacuum safe connections. The cost of vacuum rated connectors was estimated at roughly R8 000 according to the engineers at iThemba LABS. The connectors were soldered and inserted into the slotted holes. They were then glued in place to ensure that the resin will not escape through the hole. A slow curing resin was then poured into the holes, and the connector was placed in a vacuum chamber to remove all air bubbles from the resin while curing.

When the vacuum connector was first placed on the side of the scattering chamber, the chamber failed to reach the target vacuum pressure. It was found that the DB9 and DB15 connectors had minor leaks between the plastic insert and the pins connecting the connector to the wires. The coaxial connector had a major leak, and it was found that air was sucked through the cable, between the centre and the outer shields. To stop these leaks, the coaxial connectors were filled with resin, the DB9 and DB15 connectors were sprayed with liquid silicon glue and a high quality *Prestik* was placed around all the connectors. This *Prestik* is very clean with very small molecules, resulting in a much better seal than normal *Prestik*. The connector, along with the fixes, are shown in Figure 4.9.

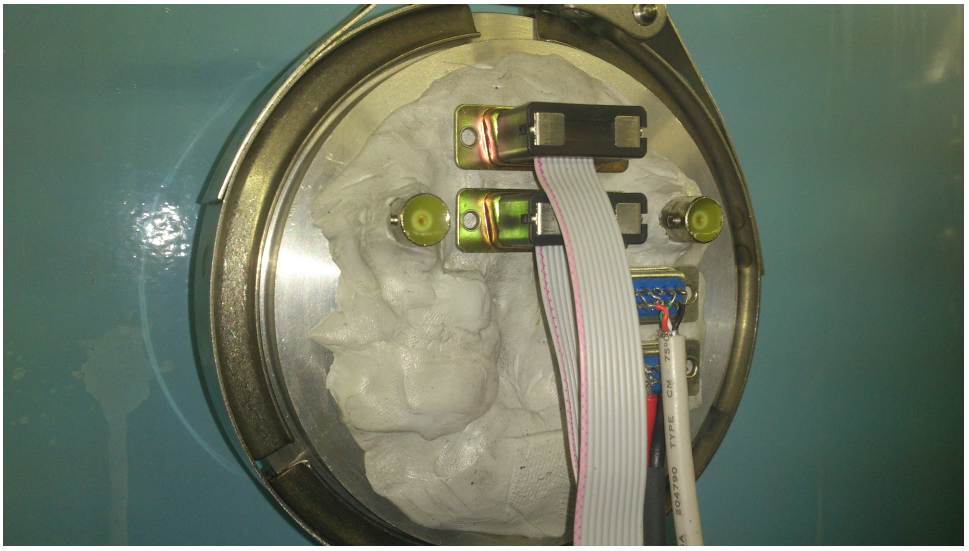

Figure 4.9 – Image of the Vacuum Connector Showing the Modifications

Connection	Proposed Connector	Description
USB	DB9	Communication with quad channel UART
Power	Coaxial	5V supply for the hardware
JTAG	2x DB15	Programming of the FPGAs

Table 4.5 – Altered List of Connections to the Hardware

4.4 Firmware Design

After the hardware was populated and tested the final firmware designs could be implemented and programmed onto the FPGAs.

4.4.1 Controller Libero Design

Figure D.2 in Appendix D shows the final Libero design for the controlling FPGA. The design consists of the Core8051s processor, connected to the external memory via the custom memory interface as described in §4.2.1. The Core8051s is also connected to the APB bus on which it is the master. Several slaves are connected to the APB bus:

1. CoreI2C

   Used for communication with the on-board sensors.

2. 2x CoreTimer

   Used to generate interrupts for time based functions, such as data sampling and UART servicing.

3. CoreWatchdog

   Used to detect faults in the Core8051s.

4. CoreInterrupt

   Used to connect multiple interrupts to the two single interrupt channels on the Core8051s.

5. 2x Original CAN Controller

   Used to communicate with the CAN Controllers on the Device Under Test, while the second is used to acknowledge all messages sent on the CAN bus.

6. 2x CoreUARTapb

   Used for communication with the computer.

7. CoreGPIO

   Used to power down the DUT, manage the CAN bus, reset the CAN controllers and toggle LEDs during diagnostics.

8. AMBAencoder

   Used to route the APB bus signals over the inter-FPGA bus to allow the CAN controllers on the DUT FPGA to connect to the APB bus.

The I2C signals coming from the CoreI2C are connected to a tri-state buffer which combines them into the I2C bus signals. The I2C bus is connected to the two temperature sensors on the board, as well as the ADC.

The GPIO signal which resets the CAN controllers are connected through an AND gate to allow both reset signals to pass through. Since the design uses an active high reset, a low

signal on either of the reset lines will result in the CAN controllers being reset. The GPIO is also used to toggle LEDs when debugging the software design.

4.4.2 DUT Libero Design

Figure D.1 in Appendix D shows the final Libero design for the Device Under Test. The DUT dummy package is used to connect unconnected signals in the design to the pins of the FPGA. Since Libero version 10, if a pin is unconnected in the top level design, the pin is removed from the pin mapping. This dummy package prevents the pin mapper from removing these pins. It also prevents any operations from randomly occurring on either SDRAM or flash due to floating connections.

The signals coming from the controlling FPGA are reconstructed into the APB bus signals. These are then manually connected to the two CAN controllers by exposing individual signals from the bus interface. Similar to the controlling FPGA, the DUT also uses a CAN encoder to connect both CAN controllers to a single transceiver.

The DUT dummy package contains no logic units and is not connected to any of the internal circuitry, and therefore does not require any protection against the effects of radiation. The CAN encoder was developed with TMR and is already protected against the effects of radiation. The AMBA decoder contains only a few multiplexors to route the outgoing data over a single 8 bit bus. It was decided not to implement any protection on the decoder as to avoid any unnecessary logic and subsequent delays being inserted on the inter-FPGA bus.

4.4.3 Controller Firmware Design

The Core8051s acts as an interface between the software running on the computer and the hardware.

The program running on the Core8051s starts by initiating all the components connected to the APB bus. After initialisation is completed, it goes into a listening state, waiting for commands from the computer. The timers are set up to produce interrupts for two functions.

The first of these functions is to service the UART buffers. For this design a buffer size of 128 bytes was used. With a baud rate of 115 200 bps, a total of 11 520 bytes can be received every second. This means that the FIFO could be filled 90 times each second. To avoid the buffer being overrun, an interrupt is generated every 5 milliseconds to service the buffer. Although the required time is 11.1 milliseconds, extra time was allocated to allow the Core8051s to process the buffer completely before the next interrupt.

The second function is the sampling of data. There are a total of eight readings taken from the board which consists of three voltage readings and three current readings, as well as two temperature readings. To avoid heavy usage on the Core8051s, the data is sent back once a second. An interrupt is generated every 125 milliseconds, at which time one of the eight sensors is read and the data transmitted over UART.

The program also features a command buffer, which contains instructions to be sent to the CAN controllers. The software running on the computer sends a list of commands and the Core8051s loads them into the buffer. After receiving a *flushBuffer* command, the Core8051s empties the buffer by performing the instructions. This is required to trigger two CAN controllers to start transmitting at the same time, to test whether the CAN controller correctly backs off when a conflict is detected.

4.5 Mitigation Scheme Implementation

In §3.5 several mitigation schemes and their possible implementation were discussed. In this section, the implementation and improvements of those techniques will be discussed.

4.5.1 Implementation of the TMR circuit

During the simulation of the TMR and recovery circuit it was found that the XOR3 gate does not function as expected. The definition of exclusive OR states that the output is true whenever the inputs differ. This was not the case for the XOR3 input gate of the Microsemi device. Instead, they followed the more generalized approached, which states that the output is true if an odd number of the inputs are true.

The correct gate that should be used is the ZOR3I gate. The ZOR3 gate outputs a logic '1' whenever all the inputs are the same. The ZOR3I is similar to the ZOR3 gate, only with an inverted output.

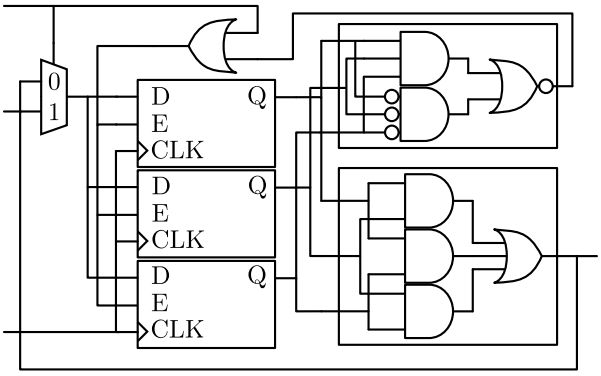

Figure 4.10 – Diagram of the TMR with Recovery Using ZOR3I

4.5.2 Improvement on the Hamming code design

After simulating the XOR3 gate it was found that it can also be used to calculate parity between the input bits. The implementation can therefore be reduced to a total of 8 gates for the encoder.

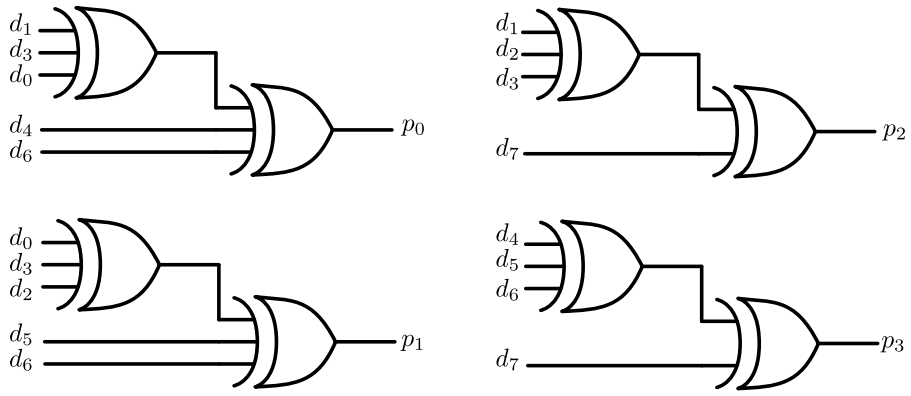

Figure 4.11 – Diagram of the Hamming Code Encoder

It also became clear that the decoding of the Hamming code could be improved on. Instead of recalculating the Hamming code parity bits and comparing them to the stored bits, the parity of each group was calculated, making the faulty bit position available immediately.

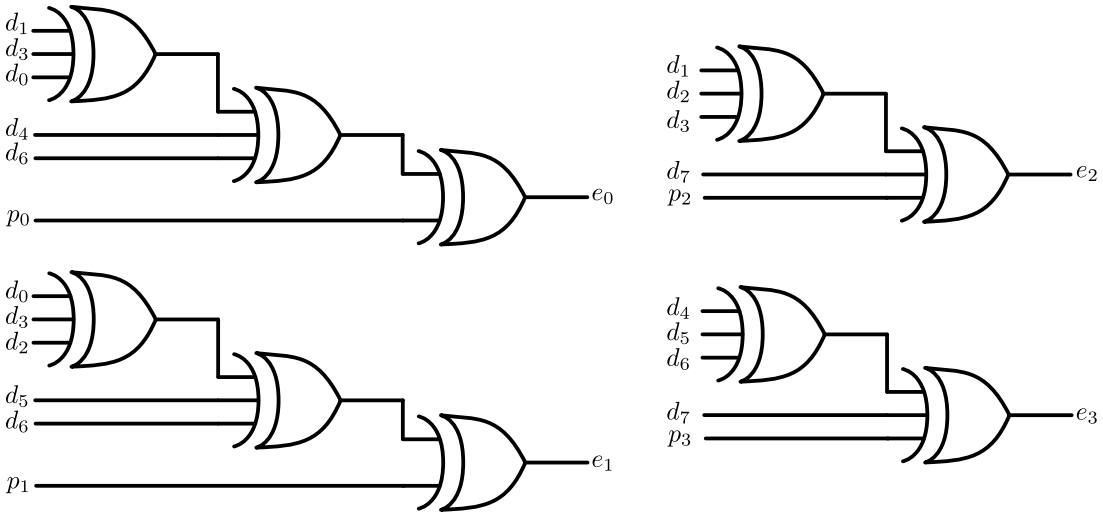

Figure 4.12 – Diagram of the Hamming Code Decoder's Wrong Bit Identification

4.5.3 SET Filter

Implementing the SET filter is as simple as using the Microsemi AND2, OR2, INV and MX2 macros to create the SET Filter.

4.5.3.1 SET Filter Component

The code from Listing 4.1 will create an SET filter component which can be inserted into the design.

```
INV INV_1_instance(.A(in), .Y(chain_0_w));
INV INV_2_instance(.A(chain_0_w), .Y(chainout));
AND2 AND2_instance(.A(in), .B(chainout), .Y(andout));
OR2 OR2_instance(.A(in), .B(chainout), .Y(orout));
MX2 MX2_instance(.A(andout), .B(orout), .S(out),
                 .Y(out));
```

Listing 4.1 – Verilog for SET Filter

The problem with this method is that it is nearly impossible to place the SET filter precisely in the design without the compiler interfering by optimizing the design. To avoid this problem, the SET filters will be injected directly into the net list post-synthesis.

4.5.3.2 SET Filter Injection

To place the SET filter exactly where it is desired without Actel's Synthesis software altering the design, a post-synthesis SET Filter injector was developed.

To insert the SET filter in the correct place, the post-synthesis design needed to be analysed. A recursive Python analysis program was developed to locate register instances and trace the input pins back to either another register instance, or to an I/O pin. The program counts the number of logic gates in series in the longest route from register input to output or I/O pin, as well as the number of logic units between the input and all the dependant outputs and I/Os.

Using the above mentioned information the SET filter can intelligently be placed with full knowledge of the effected circuit. Inserting a SET filter when there are a few gates prior to the register increases the risk of a SET occurring. Inserting a SET filter where the prior logic count is high could significantly lower the maximum operating frequency.

The SET filter injection program was written in C/C++ using QT. It is able to read the output from the Python analysis program and depending on defined parameters, insert SET filters according to the result of the analysis. The application performs a lookup of every register in the analysis, and severs the input connection. It then instantiates the required logic units to form the SET filter, as well as the required nets to connect them. From Figure 4.15 and Figure 4.16, the effect of SET injector can be confirmed.

4.5.3.3 Alteration to SET Filter Structure

Due to advances in the FPGA fabric buffer cells are no longer limited to input and output cells and clock signals, but can now also be used as part of the FPGA fabric. Where, with the previous method, the delay element size is limited to factors of two inverter gates, using buffers, we are now able to use factors of one. This not only results in a finer detection resolution of SET pulses, but also allows a better optimized curve between the SET suppression length and delays introduced.

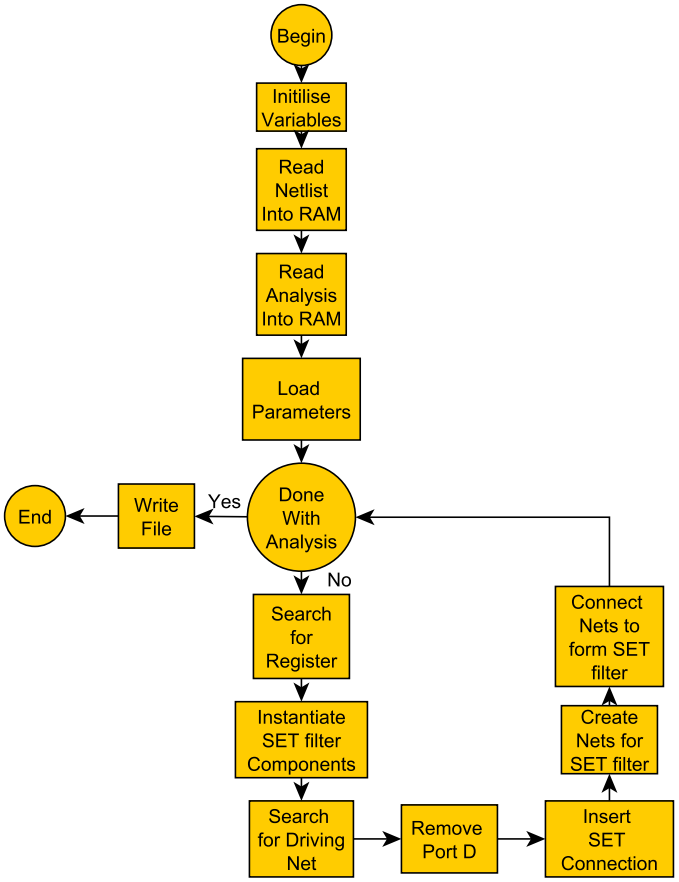

Figure 4.13 – Flow-diagram of the SET Filter Injector

Figure: 4.17 shows the number of SETs which propagate through the filter plotted against the duration of the SET pulse. For the top two graphs the blue line on the left represents a single buffer in the delay chain, while the red line on the right represents ten buffers. For the lower two graphs the blue line on the left represents two inverters in the delay chain, while the purple line on the right represents ten inverters.

The test for the buffer consisted of a set of 44 SET filters of which the delay element length is increased one at a time up to ten times, while the inverter test consisted of a set of 88 SET filters of sizes two, four, six, eight and ten. Microsemi Designer was given the freedom to place the components anywhere on the FPGA, as to simulate similar behaviour when the final design will be mitigated.

When the delay unit contains more than five elements, the results start to become distorted. This is due to the Place-and-Route process of the Microsemi Designer having to place inverters or buffers away from one another to optimize the area usage, power usage or timing to improve the device performance. While it is possible to turn these optimizations off, a well spread layout is desired for the final test along with optimum timing to ensure that there are no timing issues.

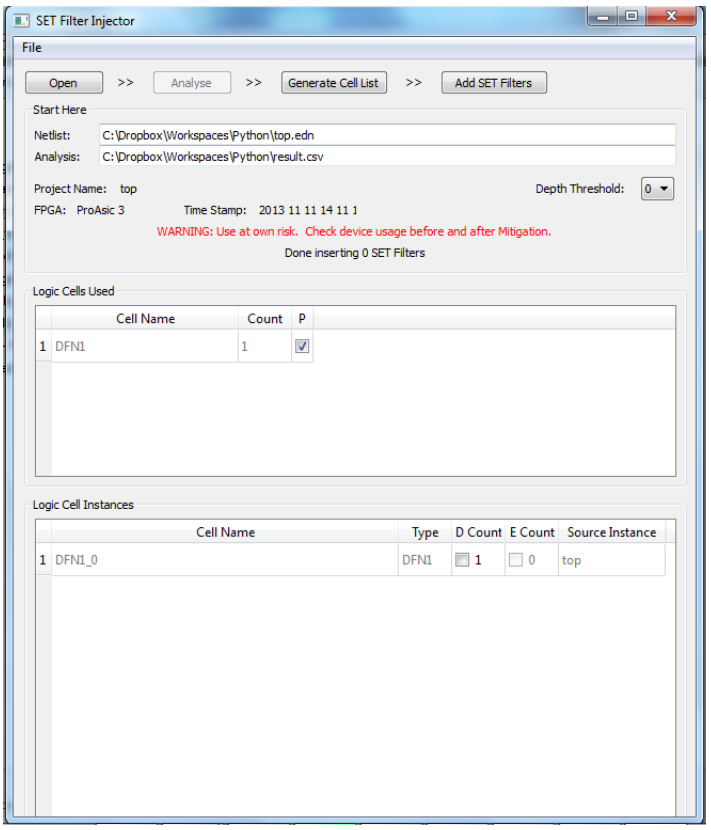

Figure 4.14 – Screen-shot of the SET Filter Injector

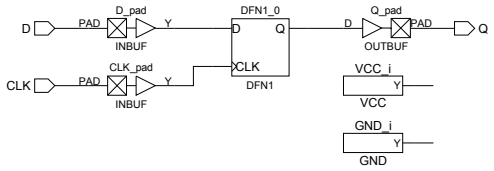

Figure 4.15 – Register Before SET Filter Injection

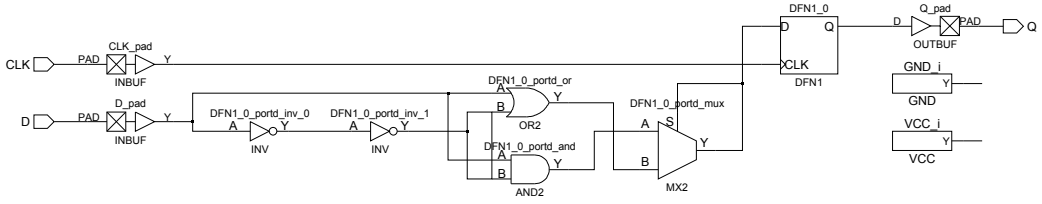

Figure 4.16 – Register after SET Filter Injection

4.5.3.4 SET Capture and delays

With both the inverter and buffer chains, a significant decrease in the maximum frequency of the CAN controller was noticed. This dencrease in maximum frequency is due to the delay of each element in the SET filter, as well as delays in the routing inside the FPGA.

To see the extent of the delays caused by different lengths of delay chains, 400 SET filters

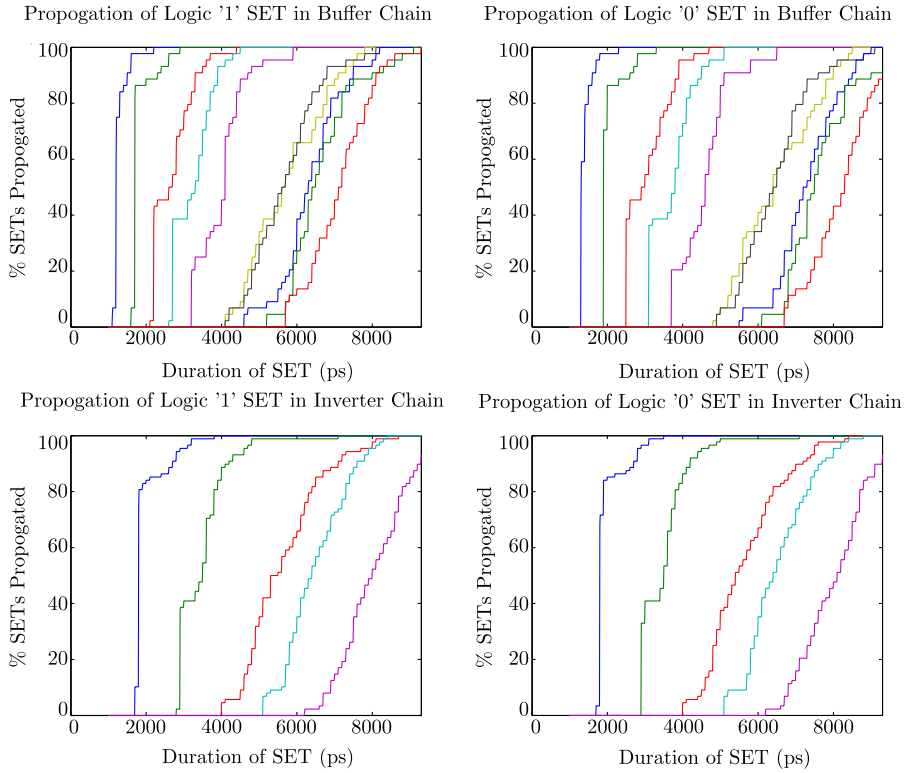

Figure 4.17 – Comparison Between Buffer and Inverter Based SET Filters of Different Lengths

were placed in a design and simulated post-layout. This will take into account both the delays in each element, as well as the delay for different routes. The same device was used for simulation, but with a different IC footprint. This allows the simulation to use more I/O pins, while the results are still applicable to the FPGA used in this design.

Figure 4.18 shows the delay of each instance of the SET filter for different delay chain lengths. At the bottom, a zero element delay chain was used as baseline to get the delay from the voting circuit, as well as input buffers and routing effects. The post-synthesis layout simulation treats the simulation as if it was running on an FPGA. This results in both input and output buffers adding to the delay, along with the routing paths from the voting logic to a valid I/O.

Table 4.6 shows the average delays of each chain, as well as the increase between different delay chain lengths.

4.5.3.5 Longest Path and Maximum Frequency

The estimated maximum frequency of the CAN controller is roughly 55 MHz, as per the synthesis results. This frequency is calculated by adding the delays of all the cells in the longest path between two memory cells, such as DFF, latches, inputs or outputs. This is referred to as a pipeline. With a maximum frequency of 55 MHz for the ProASIC A3PE1500

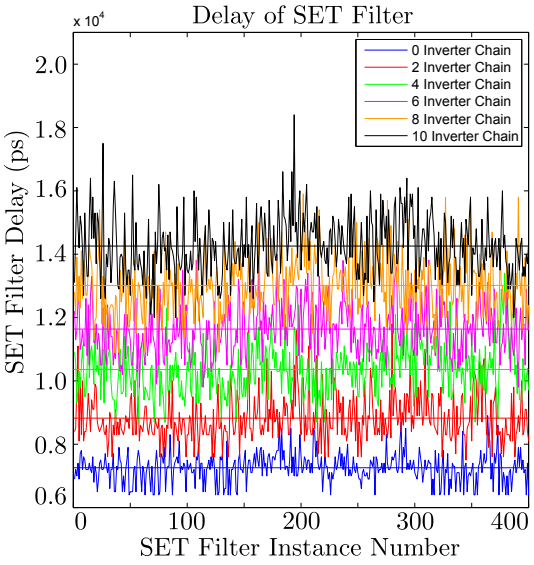

Figure 4.18 – Average Delays of SET Filters of Different Chain Lengths

device the longest pipeline is roughly 18.18 ns.

The voting logic of the SET filter adds on average 7 256 ps to the design as per Figure 4.18, while each increment of the delay chain adds an average of 1 400 ps as per Table 4.6.

If, in the worst case, each SET filter adds about 7 ns delay and about 700 ps delay per inverter to the design, then the new maximum frequency of the device can be calculated using Equation 4.5.1. The results are listed in Table: 4.7.

$$f_{max_{new}} = ({f_{max}}^{-1} + 7256ps + n \times 700ps)^{-1} \tag{4.5.1}$$

From the results summarised in Table 4.7 it is evident that the use of the SET filter severely compromises the maximum frequency of the design. With SETs reaching pulse widths of up 4 ns in documented cases [33], a total of 8 inverters are required to block the SET. While the SET filter is a novel idea, the practical application has a negative impact on the operating frequency of the design.

Inverters per SET Filter	**Average delay of SET Filter (ps)**	**Delay of Delay Chain (ps)**	**Increase (ps)**
0	7256		
2	8821	1565	1565
4	10358	3102	1537
6	11635	4379	1277
8	13012	5756	1377
10	14257	7001	1245

Table 4.6 – Average Delay for Different SET Filter Lengths

Inverters per SET Filter	New max frequency
2	37.26 MHz
4	35.42 MHz
6	33.74 MHz
8	32.22 MHz
10	30.83 MHz

Table 4.7 – Maximum Frequency of CAN controller Due to Delays inside SET Filters

4.6 CAN Controller Implementation

This section covers the implementation of alterations and mitigation techniques to the CAN controller.

4.6.1 Removal of Non-Reset Registers

After the correct interface was added in §3.4.1 it was found that running post-synthesis simulations resulted in abnormal device performance, even though it operated normally in hardware. Even after testing the 8051 and Wishbone interfaces on the same test-bench, the problem persisted. Inspection of the signals in some of the lower level peripherals showed that somewhere between the release of the reset signal to the device, and removal of the reset mode flag, internal signals went into an unknown state and was unable to exit. One of the signals that went into an unknown state, was *tx_point*.

```
1 tx_point <=#Tp ~tx_point & seg2 &
                 (  clk_en & (quant_cnt[2:0] == time_segment2)
                 | (clk_en | clk_en_q) & (resync | hard_sync) );
```

Listing 4.2 – tx_point signal assignment

From Listing 4.2 it is clear that *tx_point* is assigning itself, along with other signals, to itself. If, at any stage, any of the other signals enter an unknown state, the unknown state will remain in *tx_point*. It just so happens that some of the configuration registers have no action coupled to the reset pin. This resulted in *time_segment_2* being unknown until data is written to it.

The reason some of the registers do not have a reset action coupled to it, is because the CAN controller was designed to be a replica of the SJA1000 CAN controller developed by Phillips. Some of the configuration registers keep their value even through a full reset. Because hardware only knows a logic '1' or '0', it is unable to produce an unknown state. This is why the Verilog CAN controller worked in hardware, but post-synthesis simulation resulted in unexpected performance. The solution was to replace all configuration registers that do not have a reset action coupled to it with a register that clears its contents when reset is active.

4.6.2 Flattening of the CAN Controller

To avoid the nine times instantiation when implementing TMR on a hierarchical design as mentioned in §3.4.2, the design of the CAN controller had to be flattened out. This involved removing instantiations from the sub-modules of the CAN controller and placing them in the top level.

It initially seemed a trivial process to move registers from the registers module to the top level. The data input to the registers were all connected to the data input of the CAN controller. The write-enable signals where all generated at the top of the registers module, and could simply be changed from wires to outputs. For the registers requiring alternative reset signals to the reset input of the CAN controller, a similar process as the write-enable signals was followed.

When it came to larger components such as the FIFO and acceptance filter, however, some additional inputs and outputs had to be created or removed, making the process more complicated. In some cases, the connections to the lower level modules where connected directly to signals coming from other modules in the top level. These could be connected directly and the matching inputs and outputs from the bit stream processor could be removed. In other cases, these signals were generated inside the bit stream processor and had to be converted to output signals. In some cases, these signals were connected to registers, and in other cases it was connected to sequential logic. To maintain the same functionality, additional outputs were created and the outputs from the registers were assigned to the output ports.

4.6.3 Protecting the registers

Each register was synthesized to find exactly which logic cells were used. Based on this, the TMR was implemented in the file.

4.6.3.1 Register With Asynchronous Reset

Registers containing an asynchronous reset signal can be instantiated using the *DFN1E1C1* macro. To protect the register using TMR and implement automatic recovery, the code in Listing 4.3 was used.

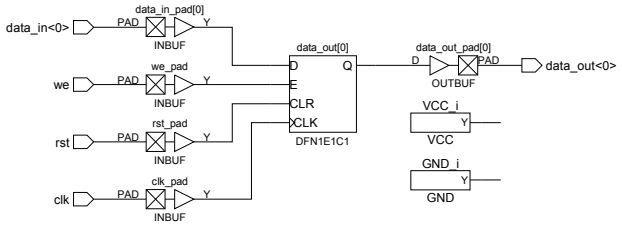

Figure 4.19 – Register Prior to TMR and Recovery

```
1  MX2 data_in_mx_0_i (.A(data_out), .B(data_in), .S(we),
2                      .Y(data_in_mx));
   DFN1E1C1 data_out_0_1_i (.D(data_in_mx), .E(reg_enable),
                      .CLK(clk), .CLR(rst), .Q(data_out_bank_1));
   DFN1E1C1 data_out_0_2_i (.D(data_in_mx), .E(reg_enable),
                      .CLK(clk), .CLR(rst), .Q(data_out_bank_2));
7  DFN1E1C1 data_out_0_3_i (.D(data_in_mx), .E(reg_enable),
                      .CLK(clk), .CLR(rst), .Q(data_out_bank_3));
   MAJ3 data_out_maj_0_i (.A(data_out_bank_1),
                      .B(data_out_bank_2),
                      .C(data_out_bank_3), .Y(data_out));
12 ZOR3I error_0_i (.A(data_out_bank_1), .B(data_out_bank_2),
                      .C(data_out_bank_3), .Y(error));
   OR2 error_enable (.A(we), .B(error), .Y(reg_enable));
```

Listing 4.3 – TMR and Recovery implemented on a single bit register

Listing 4.3 instantiates three register components as well as the required voting and repair logic, as designed in Figure 3.8. This results in the circuit in Figure 4.19 changing to the circuit in Figure 4.20.

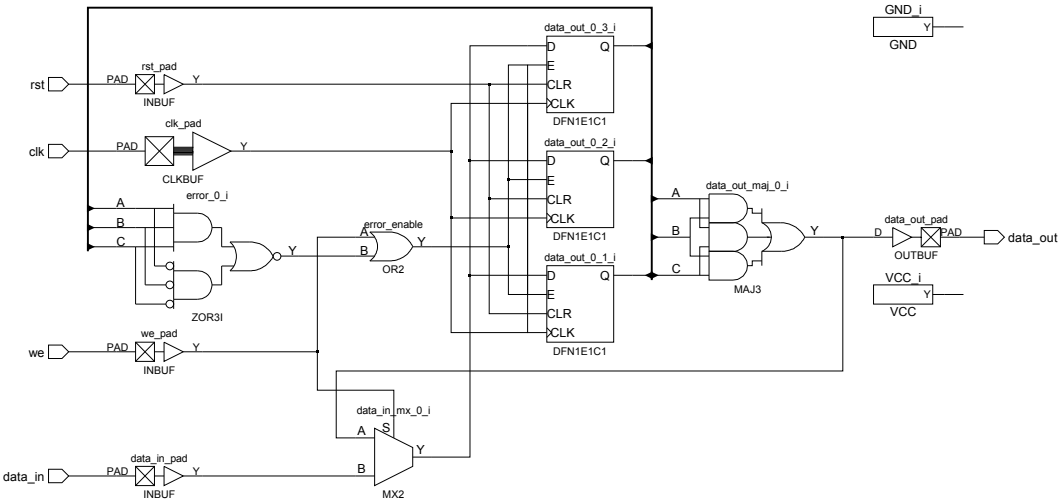

Figure 4.20 – Register After TMR and Recovery

4.6.4 Protecting the FIFO module

To implement the Hamming Code encoders and decoders, four individual blocks were created. This includes an encoder and decoder for both five and eight bit data. The blocks for five bit data is the same as that of the eight bit data, only scaled down.

The 8 bit data encoder can be created using the code in Listing 4.4.

```
XOR3 PAR_1_GEN_1_i (.A(data_in[0]), .B(data_in[1]),
                    .C(data_in[3]), .Y(e0_w));
XOR3 PAR_1_GEN_2_i (.A(e0_w), .B(data_in[4]),
                    .C(data_in[6]), .Y(hamm_out[0]));
XOR3 PAR_2_GEN_1_i (.A(data_in[0]), .B(data_in[2]),
                    .C(data_in[3]), .Y(e1_w));
XOR3 PAR_2_GEN_2_i (.A(e1_w), .B(data_in[5]),
                    .C(data_in[6]), .Y(hamm_out[1]));
XOR3 PAR_3_GEN_1_i (.A(data_in[1]), .B(data_in[2]),
                    .C(data_in[3]), .Y(e2_w));
XOR2 PAR_3_GEN_2_i (.A(e2_w), .B(data_in[7]),
                    .Y(hamm_out[2]));
XOR3 PAR_4_GEN_1_i (.A(data_in[4]), .B(data_in[5]),
                    .C(data_in[6]), .Y(e3_w));
XOR2 PAR_4_GEN_2_i (.A(e3_w), .B(data_in[7]),
                    .Y(hamm_out[3]));

assign word_out[0] = hamm_out [0];
assign word_out[1] = hamm_out [1];
assign word_out[2] = data_in [0];
assign word_out[3] = hamm_out [2];
assign word_out[4] = data_in [1];
assign word_out[5] = data_in [2];
assign word_out[6] = data_in [3];
assign word_out[7] = hamm_out [3];
assign word_out[8] = data_in [4];
assign word_out[9] = data_in [5];
assign word_out[10] = data_in [6];
assign word_out[11] = data_in [7];
```

Listing 4.4 – Hamming Code encoder for 8 bit data

The 8 bit data decoder can be created using the code in Listing 4.5. The code in Listing 4.5 only determines which bit should be flipped.

4.6.5 Protecting the CRC module

In some cases it is easier to implement the TMR directly inside the module, instead of triplicating the entire module. In the case of the CRC module, a single clock sensitive process was defined which is used to calculate the CRC code. To implement the TMR all registers where defined a second and third time. The clock sensitive process was then copied twice. The register containing the CRC code was then put through a majority voter to vote on the correct code.

```
XOR3 PAR_1_MIS_1_i      (.A(data_in[0]), .B(data_in[1]),
                .C(data_in[3]), .Y(e0_w));
XOR3 PAR_1_MIS_2_i      (.A(e0_w), .B(data_in[4]),
                .C(data_in[6]), .Y(e1_w));
XOR2 PAR_1_MIS_3_i      (.A(hamm_in[0]), .B(e1_w),
                .Y(position[0]));
XOR3 PAR_2_MIS_1_i      (.A(data_in[0]), .B(data_in[2]),
                .C(data_in[3]), .Y(e2_w));
XOR3 PAR_2_MIS_2_i      (.A(e2_w), .B(data_in[5]),
                .C(data_in[6]), .Y(e3_w));
XOR2 PAR_2_MIS_3_i      (.A(hamm_in[1]), .B(e3_w),
                .Y(position[1]));
XOR3 PAR_4_MIS_1_i      (.A(data_in[1]), .B(data_in[2]),
                .C(data_in[3]), .Y(e4_w));
XOR3 PAR_4_MIS_2_i      (.A(e4_w), .B(data_in[7]),
                .C(hamm_in[2]), .Y(position[2]));
XOR3 PAR_8_MIS_1_i      (.A(data_in[4]), .B(data_in[5]),
                .C(data_in[6]), .Y(e5_w));
XOR3 PAR_8_MIS_2_i      (.A(e5_w), .B(data_in[7]),
                .C(hamm_in[3]), .Y(position[3]));
```

Listing 4.5 – Hamming Code decoder for 8 bit data

4.6.6 Protecting the ACF module

To insert the SET Filter, the assignment to the ID_OK register should be intercepted. The conditional assignment was replaced with a single wire assignment, where the assignment to the wire was now conditional to the filter configuration.

The SET filter was inserted and a significant drop in the estimated frequency was observed. Because of this, TMR was considered as a better alternative to maintain a higher operating frequency.

Triplication of the ACF module showed an even worse estimated frequency. While this does not make any sense, it should be the safer option. There is far less sequential logic in series with the TMR circuit than with the SET filter.

While the TMR implementation showed a worse estimated frequency, from theory it should not be so. The exact cause of the discrepancy is unknown, as a visual inspection of the longest path clearly showed that it is shorter than when using the SET filter. Given the higher expected protection and comparative operating frequency TMR was chosen over an SET filter to protect the ACF module.

4.6.7 Protecting the BTL and BSP modules

To protect the BTL module using TMR, the BTL module instantiation was modified. The shell of the BTL module was kept, while inside three BTL module instances were defined. The inputs were all mapped to the input of the shell while the outputs where mapped to newly declared wires. Majority voters were then used to vote on the correct output. The same process was used for the BSP module.

4.7 Hardware Test Design

4.7.1 Controlling Software

Since the Ethernet interface was no longer implementable a UART interface was developed. It has a much lower bandwidth than the Ethernet and usage will be limited. Since two of the four UART channels are connected to the controller, one will be used to monitor the board, while the other will be used to perform tests. Both of these interfaces use the same command structure.

Byte no	Read	Write
1	';'	':'
2	Address	Address
3		Data

Table 4.8 – Command Structure of PC-to-PCB Communication

1	2	3	Description
':'	0xEC	'0'	DUT Power Off
':'	0xEC	'1'	DUT Power On
':'	0xFF	0x01	Flush Command Buffer
':'	0xFF	0x02	Release CAN Bus
':'	0xFF	0x04	Pull CAN Bus Low
':'	0xFF	0x08	Ping Request, Send Ping Back
':'	0xFF	0x10	Set Reset Signal to CAN Controller
':'	0xFF	0x20	Remove Reset Signal From CAN Controllers
':'	0x00 + ADDR	Data	Insert Write Command Into Command Buffer 0
':'	0x20 + ADDR	Data	Insert Write Command Into Command Buffer 1
':'	0x40 + ADDR	Data	Insert Write Command Into Command Buffer 2
';'	0x00 + ADDR		Insert Read Command Into Command Buffer 0
';'	0x20 + ADDR		Insert Read Command Into Command Buffer 1
';'	0x40 + ADDR		Insert Read Command Into Command Buffer 2

Table 4.9 – List of Commands of PC-to-PCB Communication

Both the PC software and board firmware used the same command structure. While the board may not request a read operation, a write operation will write the register value for the accompanying address field.

To monitor the DUT's health, the board samples the voltages, currents and temperatures every 125 ms and sends back one of the values. This transaction has a ping-like functionality, informing the user that the board is still active.

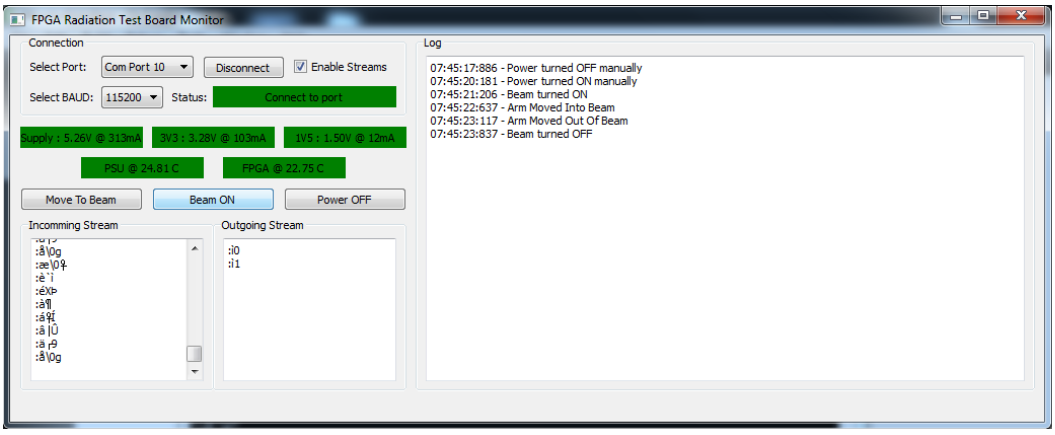

Figure 4.21 – Screen-shot of the Monitor Interface

Starting at the top left Figure 4.21, the Monitor Interface offers some configuration options. The user should select the correct communication port as well as the baud rate. The software should then be able to connect to the DUT when the user initiates the connection by pressing the *Connect* button. The *Status* field shows the status of the connection. The *Enable Streams* check box offers the user the ability to deactivate the display of the incoming and outgoing data streams. Then it shows the voltages and currents of the DUT power supply as well as the total power supply to the board. For logging purposes, there are some buttons which mimic a physical action. The *Move to Beam* button logs that the control room operator has now initiated a movement on the arm to place the board in the radiation beam. Similarly, the *Beam On* button logs that the control room operator has now turned on the radiation beam. The *Power Off* button sends a command to the board to turn off the DUT's power supply. Finally, there is a session log on the right hand side showing events with the corresponding time at which they occurred. All information in this window is logged, as well as every packet coming in or going out of the communication port.

This application serves as a template for all the testing software that will be developed.

4.7.2 Implemented Tests

More tests on a smaller scale will give a better idea of the working of each sub-component in the CAN controller. Due to the shift in focus from physical radiation testing to simulation only a few tests were designed.

4.7.3 Operational Test

The operational test simulates normal CAN bus activity. The firmware consists of four CAN controllers. The first CAN Controller is used to send and receive message on the CAN

bus, while remaining isolated from the DUT. The second CAN controller is the unmitigated version located on the DUT. The third CAN controller is the mitigated version, also located on the DUT, while the fourth CAN controller is located on the controlling FPGA. The latter is used to acknowledge all messages sent on the CAN bus.

Initially, all the devices were configured to use the same 125 kbps bus speed, with different acceptance codes and masks. This will allow each of the first three CAN controllers to have unique IDs, while the fourth accepts all messages. Thereafter, the configuration is read back to the computer and compared to the stored values. After a short delay data will be loaded into one of the CAN controllers and a transmission request will be sent. Each of the two CAN controllers in the DUT will send a message to the first CAN controller. The first CAN controller will also send a message to each of the CAN controllers on the DUT.

This process is repeated 10 times, to form a short 2 minute test. During this test, all incoming data to the communication port will be logged. A test log will also be created which contains comments on the actions taken by the testing software. Finally, an error log will log all the mismatches in the registers along with the theoretical values.

4.7.4 Configuration Hold Test

The configuration hold test follows a similar methodology as the operational test. A default configuration is loaded into the CAN controller, and is read back every second using a time-out. Whenever a register's value differs from the one stored after an initial read, an SEU has occurred. If any of the interrupt or error registers change, it is also an indication of an SEU, since all CAN controllers are idle. The test sequence can be described as follows:

1. Hard reset the CAN controllers
2. Force Reset Mode on all Can controllers
3. Load Control, ACF, BT, CLKDIV with values
4. Switch to operating mode
5. Load Transmit Buffer
6. Switch to Reset Mode
7. Switch to Extended Mode
8. Load Mode, IE, EWL, RXEC, TXEC, ACF with values
9. Perform Initial read of all registers
10. Switch on beam and move board into beam.
11. Repetitively switch between modes and read values

Due to the various delays prior to the radiation testing this test was abandoned and priority was given to the simulations and operational test.

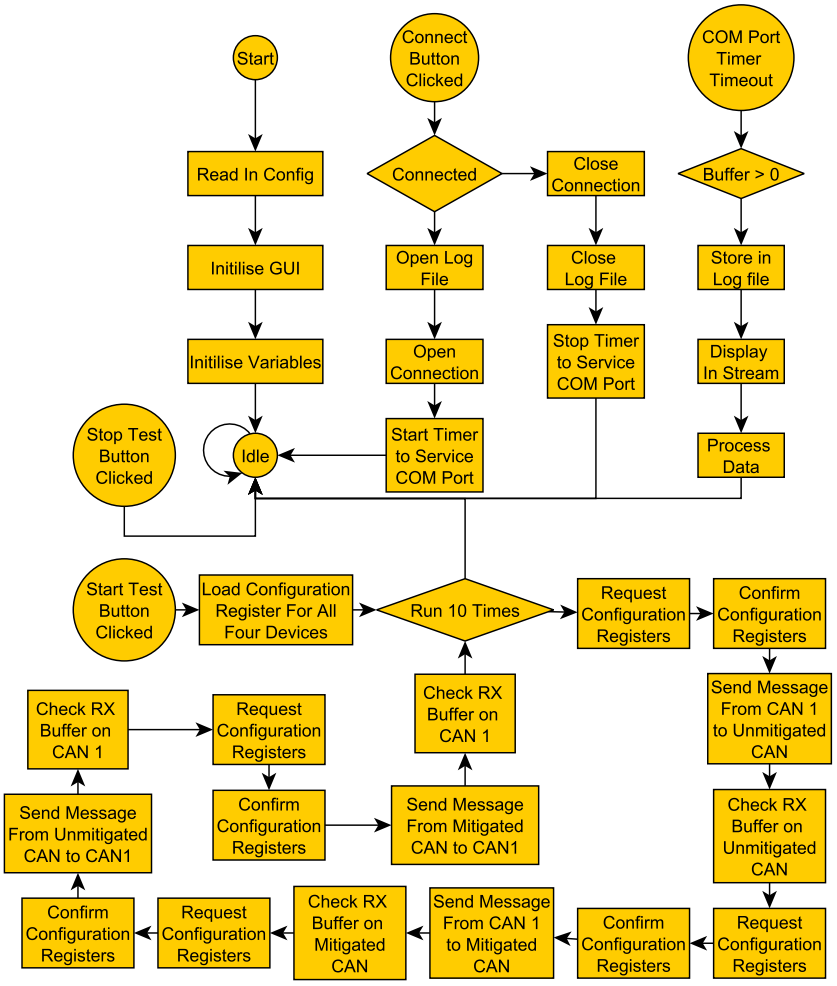

Figure 4.22 – Flow Diagram of the Operational Test

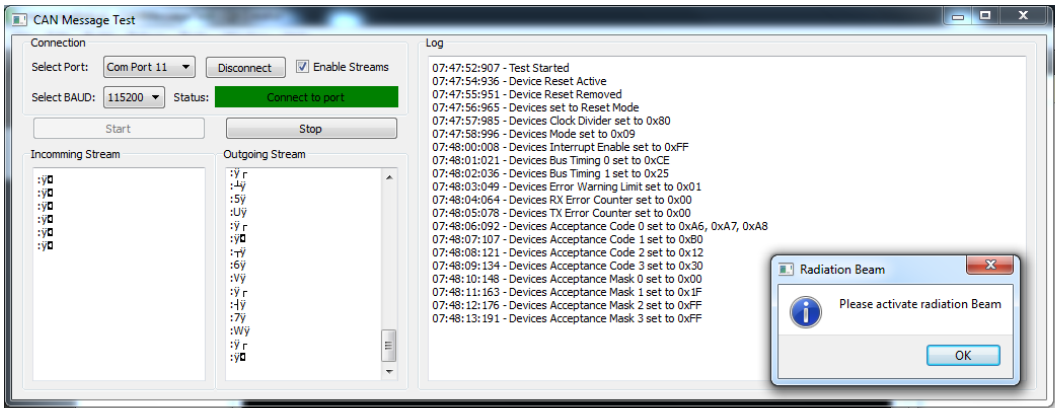

Figure 4.23 – Screen-shot of the Operational Test's Software Interface

4.8 Simulated Test Design

To verify the working of the CAN controller with the implemented mitigation techniques, the CAN controller was simulated and the mitigated areas monitored.

4.8.1 Error Detection and Correction on FIFO block

The Hamming code implemented on the FIFO block is straight forward. When a message is received on the CAN bus, it is stored in a temporary buffer. The data stays in this buffer for the duration of the CAN message.

Once reception is complete, the data is moved from the temporary buffer to the FIFO buffer. After data was written to the FIFO buffer, the information regarding the length of the message and status of the FIFO buffer is stored in a second FIFO buffer. The temporary buffer is located in the BSP, which is protected using TMR and data coming in to the FIFO buffer can be assumed correct.

Since the encoder is only 14 gates in size, the chances of an upset occurring when writing data to the FIFO buffer can be considered minute. Thus, it only needs to be verified that the decoder is working correctly, given both correct and incorrect data. To simulate this, a message of 8 bytes will be transmitted on the CAN bus. Once received, the data will be read back to the Verilog test-bench. After the data is read back, the data in the FIFO buffer will be altered. Single data and Hamming Code bits will be changed in both directions, followed by a two bit upset.

If the Hamming Code is working correctly, the data should remain intact for all cases, except when more than one bit is altered.

4.8.2 TMR and Recovery on Registers

Since TMR is basically three registers with the same input signals, not much can go wrong. A majority voter votes on the outputs of these registers, to give the correct output. When an error occurs in one of the registers, the output should remain the same.

In the case of the recovery circuitry, an error in one of the registers will trigger a re-write on all registers. Data will be fed back using a multiplexer, only if new data is not written to the registers.

To confirm that the TMR and recovery is working correctly, data will be loaded into all configuration registers of the CAN controller. The data will be read back into the test-bench a few seconds after it being written. Once the data is read back, the value of the output signal of one of the registers will be altered through the test-bench.

If the TMR and recovery circuits are working correctly, the output of the majority voter should remain the same, and the write enable signal should be triggered to all the registers.

Chapter 5

Results

In this chapter, the results from the simulations and radiation testing will be discussed. The simulations of the CAN controller focusses on the EDAC using Hamming Codes on the FIFO buffer, as well as TMR with recovery for the long term registers. The radiation testing focusses on the behaviour of the entire CAN controller, specifically looking for upsets in the available registers.

5.1 Mitigation Results

Table 5.1 shows the result of the mitigation techniques. Comparing Table 5.1 with Table 3.1, the increase in area usage and decrease in speed is clearly visible for most of the components. The area usage of the CAN controller as a whole went up from 5 191 to 11 234 logic cells, which is an increase of 116%. The total number of registers used went up from 1 497 to 2 887 which equals an increase of 93%. The increase in registers is due to the triplication of both registers and sub-modules of the CAN controller. The sequential logic cells used went up from 3 694 to 8 347, which is an increase of 126%. The increase in sequential logic cells is mainly due to the triplication of sub-modules and the addition of majority voting logic. The maximum operating frequency also went down by 7%, from 55.8 MHz to 51.9 MHz. This is due to insertion of voting logic in the longest path of the CAN controller. This decrease in operating frequency is acceptable and will not have a negative impact on the performance of the CAN controller or the processor.

5.2 Simulations

Due to continual unforeseen delays in the scheduled physical radiation test appointments at iThemba LABS outside the author's control, it was decided to do more thorough simulations to verify expected response from SETs/SEUs in the controller.

5.2.1 Error Detection and Correction on FIFO

After modification to the two FIFO buffers, each had its own Hamming code encoders and decoders. To simulate the working of these units inside the CAN controller, the original

Module name and unit count	Register count	Logic Count	I/O Count	Max Freq
mitigated_can_top	2887	8347	34	51.9 MHz
1x *can_bsp*	1041	4204	343	52.8 MHz
1x *can_acf*	3	591	123	362.2 MHz
1x *can_crc*	45	128	19	177.4 MHz
1x *can_fifo*	1398	1841	37	58.3 MHz
1x *can_btl*	93	478	36	96.8 MHz
1x *can_registers*	24	638	336	180.4 MHz
4x *ACCEPTANCE_MASK*	24	29	19	168.3 MHz
4x *ACCEPTANCE_CODE*	24	29	19	168.3 MHz
13x *TX_DATA_REG*	8	0	19	NA
1x *MODE_REG0*	3	6	6	318.9 MHz
1x *MODE_REG_BASIC*	12	15	11	180.5 MHz
1x *MODE_REG_EXT*	9	11	13	218.6 MHz
1x *IRQ_EN_REG*	24	29	19	168.3 MHz
1x *BUS_TIMING_0_REG*	24	29	19	168.3 MHz
1x *BUS_TIMING_1_REG*	24	29	19	168.3 MHz
1x *COMMAND_REG0*	1	2	6	NA
1x *COMMAND_REG1*	1	2	6	NA
1x *COMMAND_REG*	2	3	8	NA
1x *COMMAND_REG4*	1	2	6	NA
1x *ERROR_WARNING_REG*	8	0	19	NA
1x *CLOCK_DIVIDER_REG_7*	3	4	5	315.1 MHz
1x *CLOCK_DIVIDER_REG_3*	1	0	5	NA
1x *CLOCK_DIVIDER_REG_LOW*	5	0	9	NA

Table 5.1 – Breakdown of FPGA Usage of the Mitigated CAN Controller Post-Synthesis

and mitigated CAN controllers were connected in a single design. A message was then transmitted from the original CAN controller to the mitigated CAN controller. Once the message was copied into the FIFO buffer, the data was read back. To facilitate the injection errors the *run.do* file generated for the simulation was modified to change the values of the outputs of the registers.

Addr	Original Data	Changed Data	Word Bit Flip	Data Bit Flip	Parity Bit Flip	Flip
0x14	111011011011	111011011111	3	1		0 ⇒ 1
0x15	001011001001	001011011001	5	2		0 ⇒ 1
0x16	010100110001	010100100001	5	2		1 ⇒ 0
0x17	011111001011	011110001011	7	4		1 ⇒ 0
0x18	100101011011	100101011010	1		1	0 ⇒ 1
0x19	101111101010	101111101000	2		2	0 ⇒ 1
0x1A	110111110010	110101110010	8		4	1 ⇒ 0
0x1B	111100001000	111100001010	2		2	1 ⇒ 0
0x1C	000001111111	000101111110	1, 9	5	1	Both
0x1D	111011101111	111011101111				

Table 5.2 – Contents of the CAN FIFO

Figure E.1 in Appendix E shows the waveform diagram of this simulation. Since each read transaction on the AMBA bus took two clock cycles to complete, the data was forced to change one clock cycle after the address was put on the address bus. The *fifo* signal represents the word coming from the FIFO buffer. Halfway through the read operation, the change in data can be observed. The *data_out_fifo* signal represents the data coming out of the FIFO after being decoded. The Hamming code is able to correct one bit in the word. From Figure E.1 it is clear that the decoded data from the Hamming code decoder is correct whenever a single bit is flipped. At address 0x1C, two bits were flipped which resulted in the data coming from the FIFO buffer being incorrect.

The *position[x]* signals at the bottom of the simulation represent which bit in the word is incorrect. From Figure E.1 the position of the changed bit matches that in Table 5.2, except in the case of address 0x1C. Since the Hamming code can only correct 1 bit, the two incorrect bits resulted in the wrong bit being corrected. The incorrect bit was identified as bit 8 in the word.

The *bit_x_pos_match* signals represent which data bit is incorrect. These signals correctly flip the correct data bit when they are flagged as incorrect. At address 0x1C, the error in data bit 5 was not detected and resulted in the incorrect bit going out of the FIFO buffer.

Since both the data FIFO buffer and length FIFO buffer use Hamming code encoders and decoders, this proof confirms both FIFO buffers are operating correctly.

5.2.2 Triple Modular Redundancy on Registers

After mitigation of the CAN controller several registers had a triplicated design with recovery functionality. Simulation of the CAN controller's *Bus Timing 0* register can be found in Figure E.2 in Appendix E.

Initially a value of 01000011 was loaded into the *Bus Timing 0* register, and after it was written, it was read back unchanged. When another read operation was initiated, the first of the three registers was forced to a value of 01000010. The *error_bank_0_w* signal was raised, indicating that there was an error with the last bit. After combining all the error signals, the *error_combined* indicated that there was a fault in the memory. This triggered the *reg_enable* signal to go high, forcing the data to be re-written. The effect can unfortunately not be seen, as the value was forced to a 0 value by the test-bench. Although the value of the single register has changed, the output remains correct.

A third read operation was then initiated with two of the three registers being forced to 01000010. Similar to the previous case, all the error signals triggered correctly. This time, due to the majority voter voting that the incorrect signal is actually correct, it changed its output. On the next rising edge of the clock signal, the *reg_enable* signal was already high and the new value was clocked into the registers. This is verified by the third of the three registers changing its value, since it was not forced to a value. When the data was clocked into the registers the error signals immediately lowered, indicating that there was a match in the data. When the forced signals where released, all three registers have assumed the new correct value.

5.2.3 Mitigated CAN Controller

Using the developed test-bench, both the mitigated and the unmitigated CAN controllers behaved exactly the same when simulating the mitigated CAN controller along with the original CAN controller.

Figure E.3 in Appendix E shows the simulation during which both unmitigated and mitigated CAN controllers transmit and receive a message. The simulation starts off by initiating both CAN controllers to the same configuration, with the exception of the acceptance filters. First, the unmitigated CAN controller transmits a message to which the mitigated CAN controller acknowledges receipt of the message. The unmitigated CAN controller then signals an interrupt that it has completed transmission, while the mitigated CAN controller signals an interrupt that it has received a message. Comparing the transmitted message to the received one showed that the values are the same.

The same process was then repeated, with reversed roles with the same results. This shows that the mitigated CAN controller functions the same as the original CAN controller.

5.3 Radiation Testing

Since the radiation tests were done at a late stage of this project, it was limited to a single sequence of tests.

5.3.1 Pre Radiation Tests

Prior to the actual testing, two tests were run to check the functionality of everything prior to the DUT being radiated. At this time, the radiation beam was in the process of being defocussed. To do this, the beam was set to a 2 nA particle current, and directed through a crystal. Using a camera mounted above the beam entrance hole, the crystal could be monitored which would glow when radiated.

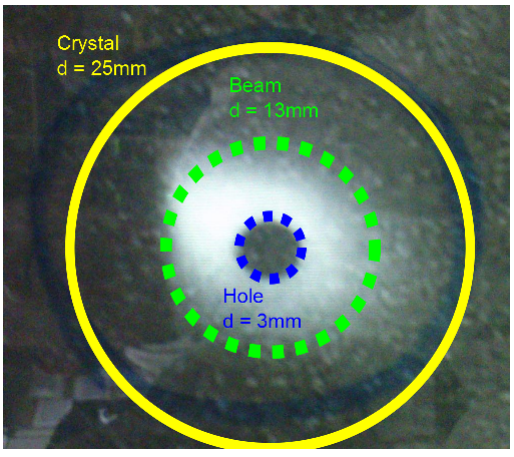

Figure 5.1 – Image of Defocussed Beam as seen in the Control Room

Figure 5.1 was taken from the monitor in the control room. The hole in the middle of the crystal has a diameter of 3 mm. The beam is estimated to be defocused to a diameter of roughly 13 mm. While this radius is not bigger than the FPGA package, it should be bigger than the die, which was estimated as roughly 5 mm x 5 mm square from a picture [74]. A later dissection of the FPGA revealed that the size is actually 8 mm x 8 mm as in Figure 5.2. Once the beam was defocused, actual testing could begin.

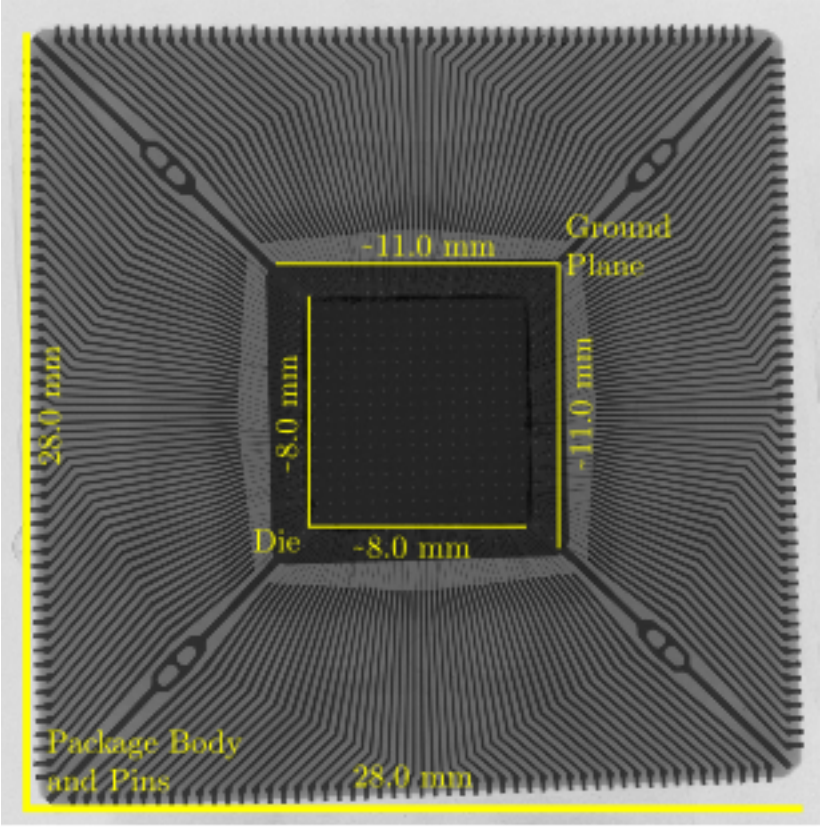

Figure 5.2 – X-Ray Image of ProASIC3 A3PE1500 FPGA Showing the Die Size

While the operators were busy defocusing the beam, the FPGA was positioned at a distance of roughly 30 cm from the beam, parallel to it. In theory, scattering of particles could influence the FPGA at this stage, but the possibility of it happening was low. When defocusing the beam during Francois' tests, only 1.18 nA was measured at the output of the vacuum chamber when applying a 2 nA beam current. The other 40% was lost due to scattering.

During the first 90 seconds of the first test, a few reading errors occurred when reading the ID 0 register from the local CAN controller. At the 90 second mark, multiple errors occurred. When inspecting the log files, it became clear that the command to put all the CAN controllers into reset had not gone through. Thus, reading back the acceptance codes and masks, returned the values of the current window in the receive buffer. A similar error occurred a few seconds later. None of these errors can be considered as being upsets, since the

DUT was not being radiated. During the second test similar errors occurred. These errors can be attributed to crosstalk on the FPGA bus, interrupts on the Core8051s overriding other interrupts, errors on the serial communication connection or memory corruption of the software running on the computer.

5.3.2 Actual Radiation Test

The required radiation test time was predetermined to roughly two minutes, as this is the period which it takes for each CAN controller to transmit and receive ten messages using the test software. For radiation testing, the initial beam current was set to 0.1 nA as this was the lowest current possible that could be delivered while staying stable to within a variation of about 10%. As with the preliminary tests, some errors were logged by the testing software that turned out not to be errors. See Table F.1 and Table F.2 in Appendix F for a log of the radiation testing.

When confirming the receive buffer for the message that was transmitted, it was found that the message sequence was incorrect. In the second test, the local CAN controller, mitigated CAN controller and unmitigated CAN controller responded with the contents of their receive buffers. The log from the communication port showed that the addresses sent from the computer, matched the incoming addresses from the CAN controllers. It can be concluded that there was some form of corruption in the test program. Because of this, most of the random errors were ignored.

Another frequent occurring error was the mismatch of the ID 1 register. This error showed up in all tests and occurred mostly in the local CAN controller. This error can be ignored, because the local CAN controller was not radiated.

During the third test errors on all three devices were noticed. Initially it was thought that one of the CAN controllers held the bus in contention or that one of the transceivers had failed. Inspection of the logs showed that the status register of each of the CAN controllers indicated that transmission was completed and that there was data in the FIFO buffers. The possibility that all three CAN controllers show the same errors is low. The only conclusion that can be made is that the controlling FPGA had been upset. For this reason, a power cycle was performed to reset the controlling FPGA.

During the fourth test a similar trend was noticed. The failures started with mixed data being logged by the test program when reading the local CAN controller's receive buffer. This was followed by errors in all the CAN controllers, with no common cause. Data varied with changes of more than four bits and in more than 30% of the cases the value zero was read.

During the fifth test functionality seemed to have returned to normal. At the beginning of the test, an upset occurred in the mode register of the unmitigated CAN controller. This upset remained in the CAN controller for the remainder of the tests. The error on the ID 0 register also returned, but in total, the errors were about 80% less than the previous tests.

For the sixth test the beam current was increased to 0.2 nA to determine if there was any change in the upset rate. A significant increase in upsets in the unmitigated CAN controller was noticed.

Register	**Init**	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**
Interrupt Enable	255	255	255	191	191	39	39	39	38	32
Bus Timing 0	199	199	199	135	199	199	199	134	134	4
Bus Timing 1	37	37	37	37	37	39	39	37	2	0
Acceptance Code 0	167	5	5	4	4	4	4	4	4	0
Acceptance Code 1	160	168	166	166	166	36	36	36	36	4
Acceptance Code 2	18	176	176	176	160	160	160	160	132	4
Acceptance Code 3	48	138	138	138	136	136	136	128	128	0
Acceptance Mask 0	0	103	103	38	38	38	38	38	38	32
Acceptance Mask 1	255	172	172	172	172	38	38	46	38	36
Acceptance Mask 2	255	183	186	186	170	162	170	170	170	36
Acceptance Mask 3	255	173	173	172	172	166	174	174	174	40
RX Error Count	0	0	0	135	133	5	5	4	4	4
TX Error Count	0	0	0	168	168	168	168	128	128	128

Table 5.3 – Unmitigated CAN Controller Register Values

Register	**Init**	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**
Interrupt Enable	255	255	255	255	255	255	255	255	255	255
Bus Timing 0	199	199	199	199	199	199	199	199	199	199
Bus Timing 1	37	37	37	37	37	37	37	37	37	37
Acceptance Code 0	168	168	168	168	168	168	168	168	168	168
Acceptance Code 1	160	160	160	160	160	160	160	160	160	160
Acceptance Code 2	18	18	18	18	18	18	18	18	18	18
Acceptance Code 3	48	048	48	48	48	48	48	48	48	48
Acceptance Mask 0	0	0	0	0	0	0	0	0	0	0
Acceptance Mask 1	255	255	255	255	255	255	255	255	255	255
Acceptance Mask 2	255	255	255	255	255	255	255	255	255	255
Acceptance Mask 3	255	255	255	255	255	255	255	255	255	255
RX Error Count	0	0	0	0	0	0	0	0	0	0
TX Error Count	0	0	0	0	0	0	0	0	0	0

Table 5.4 – Mitigated CAN Controller Register Values

Table 5.3 shows the register values for the unmitigated CAN controller after each round during the final test, while Table 5.3 shows the register values for the mitigated CAN controller. The headings, R1 to R9 represents the reading taken at the end of each round of messaging during the final test. From the table it is clear that the upset rate of the unmitigated CAN controller has increased significantly. In previous tests a single upset was noticed, but after a total of 8 minutes and 27 seconds of radiation using a 0.1 nA beam current, the increase in beam current to 0.2 nA proved fatal for the unmitigated CAN controller. A total of 102 single bit upsets were identified during the 2 minute test. Significantly, no upsets were noticed in the mitigated CAN controller. This is conclusive

proof that the use of a combination of mitigation techniques led to a more resistant CAN controller, while still maintaining a smaller footprint than TMR.

5.3.3 Post Radiation Tests

Immediately after completion of the tests, Francois attempted to reconfigured the board for his tests. Before reconfiguration started, the current supplied to the core of the DUT FPGA was around 26 mA, more than double the original 11 mA. With prior TID tests on a similar FPGA, the core current went up to 6 times the original current, before the FPGA itself started to fail. One can come to the conclusion that the device was damaged by radiation, but probably not to the point of failure.

Attempting to program the DUT FPGA failed during the erase phase. Trying to read back and verify the design after the erase failed, showing that the previous programmed design had been altered. In the data sheet of the FPGA, Actel suggests that the programming circuitry be tied to ground to avoid damage to the charge pumps. Since the board needed to be reconfigured without breaking the vacuum, the programming circuitry had to remain connected to the power source. During Francois' tests using a second PCB, the charge pumps failed between 1 and 4 minutes of radiation time.

It was decided to run a few tests to see whether the design was still functional. From the first second of the test, the mitigated CAN controller was unresponsive. All attempts at reading the registers returned 0 on the data bus. Meanwhile, the unmitigated CAN controller responded when reading the configuration registers, but seemed to have some blocks missing, as it was unable to receive a message correctly. This shows that the attempt to reprogram the device started out correctly, with the charge pumps failing somewhere during the erase process meaning that the charge pumps were merely weakened by the radiation, and failed only when stress was applied.

5.3.4 Summary

Many errors were logged during radiation testing. While the serial port logs indicated that the data was mainly intact, the only conclusion that can be made is that there was a problem with the integrity of the software running on the controlling FPGA or computer. The randomness of the errors is not indicative of a specific problem.

From the results of the first 5 tests, it is not clear that there is much difference in the upset resistance of the CAN controllers. Test 6, however, clearly showed continuous upsets in the unmitigated CAN controller, indicating that the mitigated CAN controller is more resistant to SEEs.

Chapter 6

Conclusions

In this chapter, conclusions about the results of testing will be made. Possible further work that can to be done if this project is used as the base for future projects, will also be discussed.

6.1 Overview

After thorough analysis of the requirements for both the collaborative projects, a suitable test rig was developed. Despite several challenges, the board eventually offered the required testing capabilities.

Stemming from the SET filter using GG, the SET filter as developed by Dr. Farouk Smith, offered a simple solution to filter out single event transients. Through analysis and simulation, it was found that the number of delay elements required to filter out all transients significantly slowed down the maximum operating frequency of the FPGA design.

Hamming codes proved to be an efficient way to protect large amounts of memory. A low footprint asynchronous implementation was developed which encoded and decoded the data as it was being written to or read from memory. Through simulation it was shown that the implementation worked with a minimal delay being added.

Triple modular redundancy remains the best mitigation technique to protect against the effects of radiation. A self-recovering TMR circuit was developed for registers which hold data for long periods of time. Using simulation, the recovery circuit proved to be successful in repairing faulty register values.

Using a combination of the above techniques, the Verilog implementation of the Phillips SJA1000 CAN controller was sufficiently protected from radiation. SET Filters were used to filter out single event transients entering the CAN controller on all input pins. Hamming codes were used to protect the FIFO buffer which stores incoming messages. Self-recovering triple modular redundancy was used to protect configuration registers, while normal triple modular redundancy was used to protect several crucial sub-components of the CAN controller. The logic cell usage was slightly above the 200% target set out at the beginning of the project, at 216%.

Through simulation it was proven that the self-recovering triple modular redundant registers were capable of recovering from a register upset without having any impact on the behaviour of the CAN controller. Similarly, the Hamming code implementation which was used to protect the FIFO buffer, proved successful.

Through actual testing in a radiation environment, it was found that the CAN controller was more resistant to upsets, once it was mitigated.

6.2 Further Work

Despite all the work done, there is still room for improvement. It would be possible to decrease area usage further, and implement it on a more resistant FPGA.

6.2.1 Practical Solution

To improve the viability of a CAN controller in an FPGA, it should occupy a small footprint, and offer a high level of resistance. Two solutions are suggested below.

6.2.1.1 Implement Design on a Smaller IC

The M1/AGL1000 FPGA is available on the FG144 package, offering 13 824 logic cells on a 13mm x 13mm BGA. It is the smallest FPGA available that could host a mitigated CAN controller.

This FPGA is Cortex-M1 enabled and has enough extra logic to implement a soft-core processor or at least a small state machine. Together with the transceiver, the footprint will be quite small and could actually become a viable solution for use inside a satellite.

6.2.1.2 Implement Design on a Radiation Hardened IC

Microsemi also offers a range of radiation hardened FPGAs [75] which have a high resistance to SEEs. These have an SEU rate of less than 1E-10 errors per bit per day. These readings are for worst case in geostationary orbit satellites.

The Microsemi RTAX2000S/SL FPGA has a footprint of roughly 32.5 mm by 32.5 mm in a BGA package. It offers 10 752 register elements and 21 504 combinational elements.

While more than four times the area of the smaller IC suggested previously, the extra resistance against SEEs may be required.

6.2.2 Optimization of design

To fit the design onto the smallest FPGA, the design of the CAN controller can further be optimized. Some components can be removed and some altered to create a more efficient design capable of running on a smaller FPGA that consumes less power.

6.2.2.1 Internal SRAM with EDAC

Both the M1/AGL1000 and the RTAX2000S/SL have more than 50 kbit of internal RAM. It would be possible, after some modification, to use the internal RAM modules instead of implementing the register elements in the FPGA fabric. Not only will this speed up the design, but the fixed width of these modules make it ideal for implementing EDAC. While Hamming codes were used in this project, other encoding schemes such as Reed-Solomon or Convolutional code could also be investigated.

6.2.2.2 Decrease size of the Information FIFO

While the data FIFO is 64 bytes in size, the 64 byte size of the length FIFO is unnecessary. For every CAN message, the message ID and all the data is stored. The message header will always consist of two header bytes or more, containing the ID, RTR and Data bytes count. Therefore, the data buffer should be at least twice the size of the length buffer. It is therefore possible to decrease the size of the length FIFO, should a smaller design be required.

6.2.2.3 Removal of External Clock Generator

For the SJA1000 CAN controller as a physical IC, the external clock could be used for slow speed peripherals requiring a clock signal. For the FPGA implementation however, this is not required. Most FPGAs have access to a Phase Locked Loop (PLL), used to condition the clock signal input to the FPGA. These PLLs often have more than one output that can be configured at different frequencies. Even if there is no PLL present, building a counter to scale down the input clock is a trivial task. As the external clock output from the CAN controller is not used internally, it can safely be removed, reducing the required area for the CAN controller.

6.2.3 Alterations to FPGA Radiation Test Board

Should the hardware design of this project be used as the basis for similar future projects, certain alterations should be made to the design.

6.2.3.1 Use of Cortex-M1 Enabled Device

During the design of the FPGA radiation test board, it was chosen to use a Cortex-M1 enabled FPGA to simplify the design by making a processor available which does not use logic cells. Since the FPGAs used were not Cortex-M1 enabled, the processor could not be used. Although a solution has been created, it is inefficient and use of the Cortex-M1 is strongly suggested.

The Core8051s only uses one bank of the flash and SRAM ICs added to the board, and does not use the full amount of memory available. It also inserts a wait state when reading from the memory, decreasing the operating frequency of the Core8051s. Since the Core8051s

does not have access to the AHB bus, the use of the Ethernet controller proved hard to accomplish.

These drawbacks should prove that it is of the utmost importance that the FPGAs, when ordered, are indeed Cortex-M1 enabled.

6.2.4 Minor alterations to the board

While the board was able to function correctly, some minor changes to the design are suggested.

1. Pull reset line high to the Controller 3.3V line, instead of to the DUT.

   This was a simple design error. When the controller turns the DUT off, the reset line is also pulled low, resetting the controller.

2. Separate the write, read, and chip select lines to each memory bank.

   Should the Core8051s be used in future designs, having two separate memory banks could be beneficial. It would allow an AHB bus to be inserted with the Core10100APB as its master, allowing the Ethernet controller access to SRAM memory.

3. Use a four layer PCB instead of two.

   The use of a multilayer board could help with the crosstalk problems experienced when interfacing between the two FPGAs. It could also help decrease the size of the PCB by allowing more optimized routing.

4. Add current measuring to the controller PSU

   An increase in the 1.5 V current measurements will give an indication whether the controlling FPGA has been negatively affected by the radiation.

5. Add voltage dividers to all voltages going to the ADC

   The ADC is only able to sample relative to its supply voltage. With the current design, the ADC draws power from the 3.3 V line, while trying to sample 3.3 V from the DUT PSU. This results in the ADC saturating.

6.2.5 JTAG Chain and Programming Safety

From the data sheet of the Actel range of FPGAs and from results of tests in this project, it is clear that the charge pump used to erase the device is a volatile component of the FPGA. Actel suggests connecting the supply pins of the charge pumps to ground when the FPGA is not being programmed. To prevent damage to the device, the supply to the charge pumps should be configurable.

It would also be beneficial to create a JTAG chain between the devices. This would allow a single programmer to be used through a single connection to the testing chamber. To ensure that programming succeeds, the controlling FPGA should be connected last in the

JTAG chain. When using Actel's FlashPro software to program a chain of FPGAs, the last FPGA is programmed first. Should programming of a device fail, FlashPro will stop the entire process and slave devices will not be programmed.

When using a chain between FPGAs to program, the supply can easily be connected to ground by inserting a dummy connector into the external connection. If programming is required, external power can be used to power the charge pumps. This will probably protect the charge pumps from being damaged by radiation.

6.2.6 Alterations to mitigation scheme applied to the CAN controller

After radiation testing was completed, an issue surrounding the mitigation scheme applied was raised.

6.2.6.1 Protection of the command register

During the writing of the thesis, the issue surrounding the protection of the command register was raised. While commands are cleared shortly after being written, an upset is able to trigger a command. The command register is only cleared if a command has been written to the register.

To avoid an upset from triggering a command, the register should be duplicated, and the output of the two registers should be combined with an AND gate. The likelihood of two registers experiencing the same upset is unlikely. If a command is being written to the command register, both registers would contain a logic '1'. The AND gate will ensure that the command only passes through when a command has been written to the register. An XOR gate can also be used in combination with an OR gate to clear the command

Bibliography

[1] NASA: Space Math - The Deadly Van Allen Belts. Available at: `http://spacemath.gsfc.nasa.gov/weekly/3Page7.pdf`, [2013, November 18], .

[2] Balasubramanian, A., Bhuva, B.L., Black, J.D. and Massengill, L.W.: RHBD Techniques for Mitigating Effects of Single-Event Hits Using Guard-Gates. *Nuclear Science, IEEE Transactions*, vol. 52, no. 6, pp. 2531–2535, December 2005.

[3] Smith, F.: A new methodology for single event transient suppression in flash FPGAs. *ELSEVIER Microprocessors and Microsystems*, vol. 37, no. 3, pp. 313–318, May 2013.

[4] Bark, R.: Radioactive Beam Proposal for iThemba LABS. Available at: `http://www.tlabs.ac.za/pdf/RIB/RobWorkshop2011.pdf`, [2013, November 27], 2011.

[5] Mohor, I.: CAN Protocol Controller. Available at: `http://opencores.org/project,can`, [2013, November 06], March 2009.

[6] Herveille, R.: Wishbone B3: WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. Tech. Rep., OpenCores, 2010.

[7] Mostert, S.: Sumbandilasat - An operational technology demonstrator. *Acta Astronautica*, vol. 63, pp. 1273–1282, December 2008.

[8] Surrey Satellite Technology US LLC: SSTL-300 Satellite Platform. Available at: `http://rsdo.gsfc.nasa.gov/images/catalog2010/SSTL300.pdf`, [2013, November 18].

[9] MIT Satellite Team: Avionics. Available at: `http://satellite.mit.edu/avionics.php`, [2013, November 18].

[10] ISIS: CubeSatShop. Available at: `http://www.cubesatshop.com`, [2013, November 18], .

[11] ISIS: CubeSatShop - ISIS VHF downlink / UHF uplink Full Duplex Transceiver. Available at: `http://www.cubesatshop.com/index.php?page=shop.product_details&flypage=flypage.tpl&product_id=73&category_id=5&keyword=i2c&option=com_virtuemart&Itemid=67`, [2013, November 18], .

[12] ISIS: CubeSatShop - CubeComputer. Available at: `http://www.cubesatshop.com/index.php?page=shop.product_details&flypage=flypage.tpl&product_`

id=106&category_id=8&keyword=i2c&option=com_virtuemart&Itemid=75, [2013, November 18], .

[13] ISIS: CubeSatShop - CubeSense. Available at: http://www.cubesatshop.com/index.php?page=shop.product_details&flypage=flypage.tpl&product_id=107&category_id=7&keyword=i2c&option=com_virtuemart&Itemid=69, [2013, November 18], .

[14] SunSpace: Sumbandilasat Mission Blog. Available at: http://sumbandilamission.blogspot.com/, [2013, November 18], March 2010.

[15] NASA: The Atmospheric Filter. Available at: http://er.jsc.nasa.gov/seh/Space_Astronomy_Teacher_Guide_Part_2.pdf, [2013, October 17], .

[16] NASA: Space Based Astronomy. Available at: http://er.jsc.nasa.gov/seh/Space_Astronomy_Teacher_Guide_Part_1.pdf, [2013, October 17], .

[17] Sharma: *Atomic and Nuclear Physics*. Pearson Education India, 2008.

[18] Ackermann, M.: Detection of Characteristic Pion-Decay Signature in Supernova Remnants. *American Association of the Advancement of Science*, vol. 339, pp. 807–811, 2013.

[19] Nerlich, S.: Astronomy Without A Telescope - Oh-My-God Particles. June 2011.

[20] Bothmer, V.: Solar Corona, Solar Wind Structure and Solar Particle Events. Available at: http://www.esa-spaceweather.net/spweather/workshops/proceedings_w1/SESSION3/bothmer_solar.pdf, [2013, November 18], August 2005.

[21] Reames, D.V.: Solar Energetic Particle Variations. Tech. Rep., NASA, 2003.

[22] Peterson, E.: *Single Event Effects in Aerospace*. Wiley, 2011.

[23] Holbert, K.E.: Radiation Effects and Damage. Available at: http://holbert.faculty.asu.edu/eee560/RadiationEffectsDamage.pdf, [2013, November 21].

[24] CERN: Radiation Hardness Assurance. Available at: http://lhcb-elec.web.cern.ch/lhcb-elec/html/radiation_hardness.htm, [2013, November 21], 2011.

[25] Leppälä, K. & Verkasalo, R.: Protection of Instrument Control Computers against Soft and Hard Errors and Cosmic Ray Effects. Tech. Rep., Technical Research Centre of Finland, 1989.

[26] Neudeck, P.G.: Silicon Carbide Technology. Available at: http://www.grc.nasa.gov/WWW/SiC/publications/CRCChapter2ndEd.pdf, [2013, November 27], October 2006.

[27] Danchenko, V.: Radiation hardening of MOS devices by boron. Available at: http://adsabs.harvard.edu/abs/1975nasa.reptQ....D, [2013, November 27].

[28] MIT: The Basics of Boron Neutron Capture Therapy. Available at: `http://web.mit.edu/nrl/www/bnct/info/description/description.html`, [2013, November 27].

[29] Sinclair, R.: High Speed, Radiation Hard MRAM Buffer. Available at: `http://www.nve.com/Downloads/sinclair-11-02.pdf`, [2013, November 27], November 2002.

[30] Wikipedia: Submarine Command System. Available at: `http://en.wikipedia.org/wiki/Submarine_Command_System`, [2013, November 21].

[31] Teifel, J.: Self-Voting Dual-Modular-Redundancy Circuits for Single-Event-Transient Mitigation. *IEEE Transactions on Nuclear Science*, vol. 55, pp. 3435–3439, 2008.

[32] Rezgui, S., Wang, J., Tung, E.C., Cronquist, B. and McCollum, J.: New Methodologies for SET Characterization and Mitigation in Flash-Based FPGAs. *Nuclear Science, IEEE Transactions*, vol. 54, no. 6, pp. 2512–2524, December 2007.

[33] Rezgui, S.: Configuration and Routing Effects on the SET Propagation in Flash-Based FPGAs. *IEEE Transactions on Nuclear Science*, vol. 55, pp. 3328–3335, 2008.

[34] IBM: Fault Tolerance Decision in DRAM Applications. Tech. Rep., International Business Machines Corp, 1997.

[35] Palsetia, D.: Error Detection and Correction. Available at: `http://www.cis.upenn.edu/~palsetia/cit595s08/Lectures08/ErrorCD.pdf`, [2013, October 14], 2008.

[36] Downey, T.: Calculating the Hamming Code. Available at: `http://users.cis.fiu.edu/~downeyt/cop3402/hamming.html`, [2013, September 08], 2000.

[37] Fiedler, J.: Hamming Codes. Available at: `http://orion.math.iastate.edu/linglong/Math690F04/HammingCodes.pdf`, [2013, September 08], 2004.

[38] European Space Agency: Space Introduction. Available at: `http://www.spacewire.esa.int/content/Home/HomeIntro.php`, [2013, November 21].

[39] European Cooperation for Space Standardization: Space Engineering - SpaceWire - Links, Nodes, Routers, and Networks. Available at: `http://snebulos.mit.edu/projects/reference/NASA-Generic/ECSS-E-40-12A.pdf`, [2013, November 21], January 2003.

[40] Maxim Integrated: Determining Clock Accuracy Requirements for UART Communications. Available at: `http://www.maximintegrated.com/app-notes/index.mvp/id/2141`, [2013, November 21], August 2003.

[41] Schwerdtfeger, M.: SPI - Serial Peripheral Interface. Available at: `http://www.mct.net/faq/spi.html`, [2013, November 21], 2006.

[42] NXP Semiconductors: I2C-bus specification and user manual. Available at: `http://www.nxp.com/documents/other/UM10204_v5.pdf`, [2013, November 20], October 2012.

[43] CiA: CAN History. Available at: `http://www.can-cia.de/index.php?id=161`, [2013, November 20].

[44] Microsemi: Neutron-induced single event upset (seu) faq. Online, August 2011.

[45] Mountain, R.: FPGA IRRADIATION @ NPTC-MGH. Available at: `http://indico.cern.ch/getFile.py/access?contribId=3&resId=1&materialId=slides&confId=140340`, [2013, November 20], July 2011.

[46] Electronics Weekly: Is Flash FPGA A Good Idea. Available at: `http://www.electronicsweekly.com/mannerisms/markets/is-flash-fpga-a-good-idea-2009-10/`, [2013, November 21], October 2009.

[47] Microsemi Corporation: ProASIC3 FPGA Overview. Available at: `http://www.microsemi.com/products/fpga-soc/fpga/proasic3-overview`, [2013, November 21], 2013.

[48] Microsemi Corporation: Fusion Mixed Signal FPGAs. Available at: `http://www.microsemi.com/products/fpga-soc/fpga/fusion`, [2013, November 21], 2013.

[49] Microsemi Corporation: SmarFFPGAs SoC FPGAs. Available at: `http://www.microsemi.com/products/fpga-soc/soc-fpga/smartfusion`, [2013, November 21], 2013.

[50] OpenCores: Soc Interconnection: Wishbone. Available at: `http://opencores.org/opencores,wishbone`, [2013, October 24], June 2010.

[51] Herveille, R.: Combining WISHBONE Interface Signals. Tech. Rep., OpenCores, 2001.

[52] Intel: *MCS-51 Family of Single Chip Microcomputers User's Manual*. Intel Corporation, 1981.

[53] ARM: AMBA Test Interface Driver. Available at: `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0046c/DDI0046_amba_ti_driver_ds.pdf`, [2013, November 20], 1996.

[54] ARM: AMBA Specification (Rev 2.0). Tech. Rep., ARM, 1999.

[55] LaBel, K.A.: Trade Space Involved with Single Event Upset (SEU) and Transient (SET) handling of Field Programmable Gate Array (FPGA) Based Systems. In: *NASA*. 2006.

[56] iThemba LABS: Accelerators Overview. Available at: `http://www.tlabs.ac.za/accel.htm`, [2013, November 21].

[57] F.Sturesson: Total Ionizing Dose (TID) Testing. Available at: `http://space.epfl.ch/webdav/site/space/shared/industry_media/04%20TID%20Testing%20F.Sturesson.pdf`, [2013, November 21], June 2009.

[58] F.Sturesson: Single Event Effect (SEE) Testing. Available at: `http://space.epfl.ch/webdav/site/space/shared/industry_media/08%20SEE%20testing%20%20F.Sturesson.pdf`, [2013, November 21], June 2009.

[59] Gaisler, J.: LEON SPARC Processor The past, present and future. Available at: `http://ramp.eecs.berkeley.edu/Publications/LEON3%20SPARC%20Processor,%20The%20Past%20Present%20and%20Future.pdf`, [2013, October 27].

[60] Young, R.: NASA GUIDELINES FOR BALL GRID ARRAY SELECTION AND USE. Available at: `https://nepp.nasa.gov/files/16205/05_014a%20Strickland%20Report%201%20BGA%20Guidelines%20090706.doc`, [2013, October 27], 06 2011.

[61] Microsemi Corporation: ProASIC3E Flash Family FPGAs. Available at: `http://www.microsemi.com/document-portal/doc_download/130701-proasic3e-flash-family-fpgas-datasheet`, [2013, February 25]. A3PE1500.

[62] Texas Instruments: 3.3-V CAN TRANSCEIVERS. Available at: `http://www.ti.com/lit/ds/slls557f/slls557f.pdf`, [2013, February 25], . SN65HVD233.

[63] AMIC Technology Corporation: 2M x 16 Bit x 4 Banks Synchronous DRAM. Available at: `http://www.amictechnology.com/datasheets/A43L3616A.pdf`, [2013, February 25]. A43L4616AV7F.

[64] Micron Technology Incorporated: Parallel NOR Flash Embedded Memory. Available at: `http://www.micron.com/~/media/Documents/Products/Data%20Sheet/NOR%20Flash/Serial%20NOR/M25P/M25P32.pdf`, [2013, February 25], . M25P32-VMF6P.

[65] Integrated Device Technology Incorporated: 3.3V CMOS Static RAM. Available at: `http://www.micron.com/~/media/Documents/Products/Data%20Sheet/DRAM/128mb_x32_sdram.pdf`, [2013, February 25]. IDT71V416L10PHGI.

[66] Micron Technology Incorporated: Parallel NOR Flash Embedded Memory. Available at: `http://www.micron.com/~/media/Documents/Products/Data%20Sheet/NOR%20Flash/Parallel/M29W/M29W160E.pdf`, [2013, February 25], . M29W160EB70N6F.

[67] Texas Instruments: DP83848C PHYTER Commercial Temperature Single Port 10/100 Mb/s Ethernet Physical Layer Transceiver. Available at: `http://www.ti.com/lit/ds/symlink/dp83848c.pdf`, [2013, March 5], . DP83848C.

[68] Texas Instruments: High-Side Measurement CURRENT SHUNT MONITOR. Available at: `http://www.ti.com/lit/ds/symlink/ina169.pdf`, [2013, February 25], . INA169.

[69] Maxim Integrated Products: 2.7V to 3.6V and 4.5V to 5.5V, Low-Power, 4-/12-Channel 2-Wire Serial 8-Bit ADCs. Available at: `http://datasheets.maximintegrated.com/en/ds/MAX1036-MAX1039M.pdf`, [2013, February 25]. MAX1038EEE+.

[70] Atmel Corporation: 9- to 12-bit Selectable, 0.5^{o}C Accurate Digital Temperature Sensor. Available at: `http://www.atmel.com/Images/Atmel-8748-DTS-AT30TS75-Datasheet.pdf`, [2013, February 25]. AT30TS75-SS8-B.

[71] Yaselli, I.: Report on the Effect of Radiation on Resistors and Capacitors for the HV filter circuit of the Endcap VPT. Tech. Rep., Department of Electronic and Computing Engineering Brunel University, 2004.

[72] Mentor Graphics Corporation: ModelSim SE Reference Manual. Available at: `https://ece.uwaterloo.ca/~ece327/protected/modelsim/htmldocs/modelsim_se_ref/nsmgchelp.htm`, [2013, October 14], 2011.

[73] Future Technology Devices International Limited: Quad High Speed USB to Multipurpose UART/MPSSE IC. Available at: `http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT4232H.pdf`, [2013, July 5]. FT4232H.

[74] Actel: Radiation-Tolerant ProASIC3 FPGAs Radiation Effects. Available at: `www.microsemi.com/document-portal/doc_download/131374-radiation-tolerant-proasic3-fpgas-radiation-effects-report`, [2013, November 25], April 2010.

[75] Microsemi Corporation: RTAX-S/SL FPGAs. Available at: `http://www.microsemi.com/products/fpga-soc/radtolerant-fpgas/rtax-s-sl#overview`, [2013, October 14], 2013.

Appendices

Appendix A

Work Distribution

As part of the combined effort in implementing the hardware, the work load was distributed as in Table A.1.

Task	**Party**
List initial hardware requirements	Mr. E.J. Thesnaar
List additional hardware requirements for his project	Mr. F.A. Nolte
Layout of PCB	Mr. E.J. Thesnaar
Manufacture of PCB	Trax Interconnect
Populating of PCB	Mr. J. Arendse
Testing of most of first PCB during population	Mr. E.J. Thesnaar
Design and implementation of Libero project for controlling FPGA, including memory interface	Mr. E.J. Thesnaar
Development of the programming sprite	Mr. E.J. Thesnaar
Testing of additional hardware for his project	Mr. F.A. Nolte
Testing of second PCB during population	Mr. F.A. Nolte
Design and manufacture of vacuum connector	Mr. A. Barnard
Design and testing of testing environment set-up	Mr. E.J. Thesnaar

Table A.1 – Distribution of tasks

Appendix B

PCB Schematics

This page has been left blank intentionally.

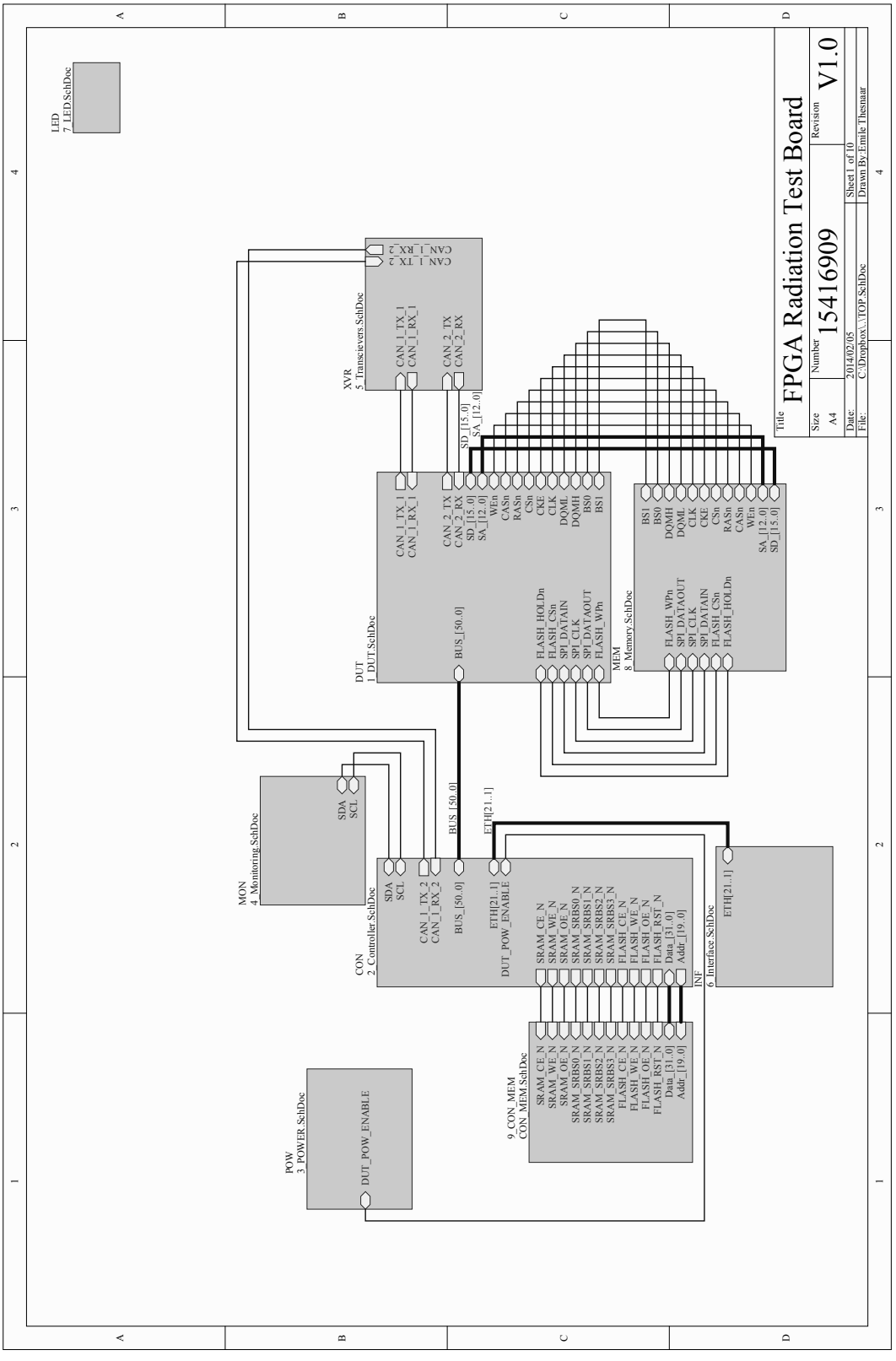

Figure B.1 – Connection of Sub-Parts of the Schematics

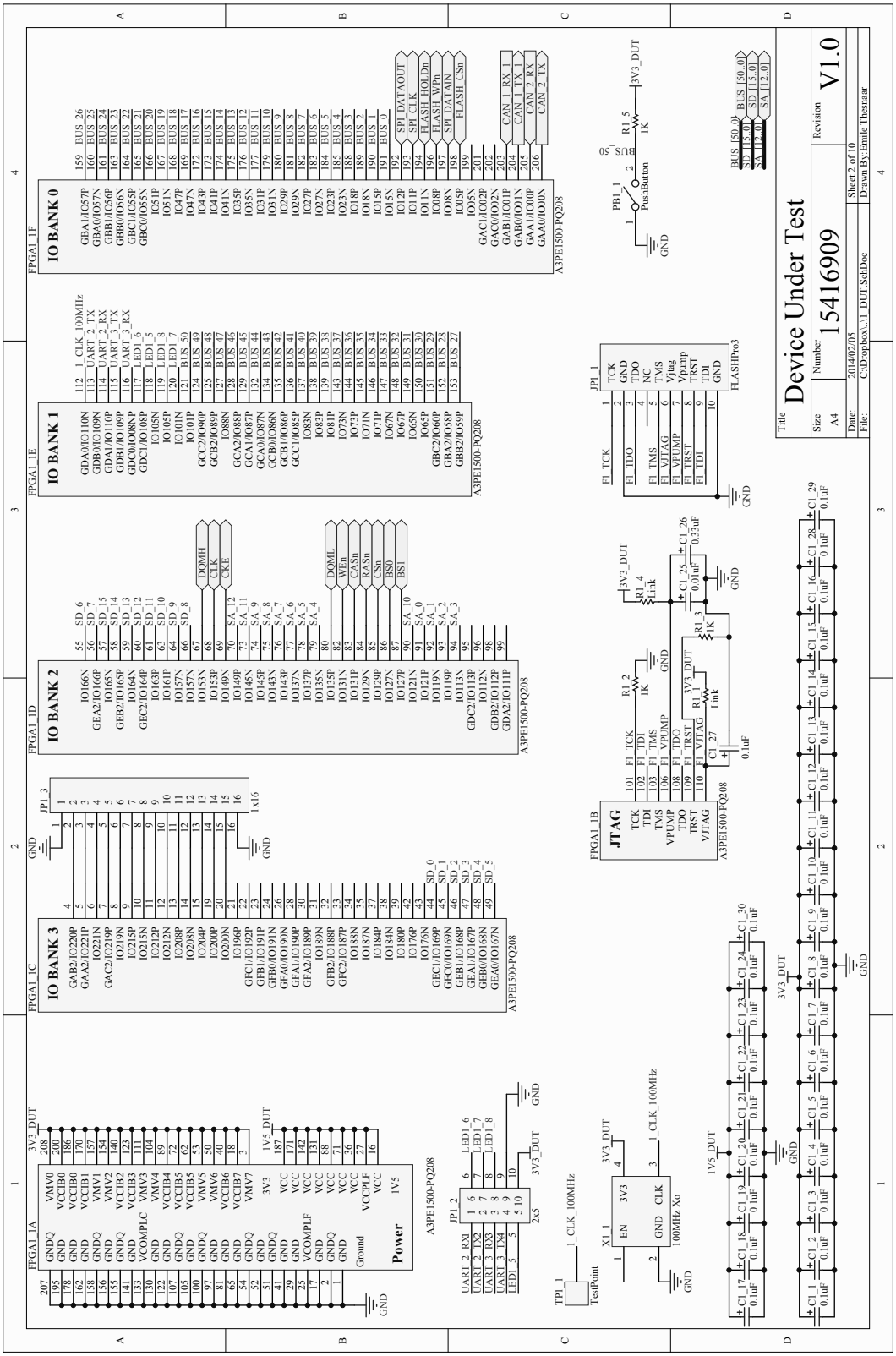

Figure B.2 – Schematic of the Device Under Test FPGA showing I/O Connections, Crystal, Decoupling Capacitor and Headers

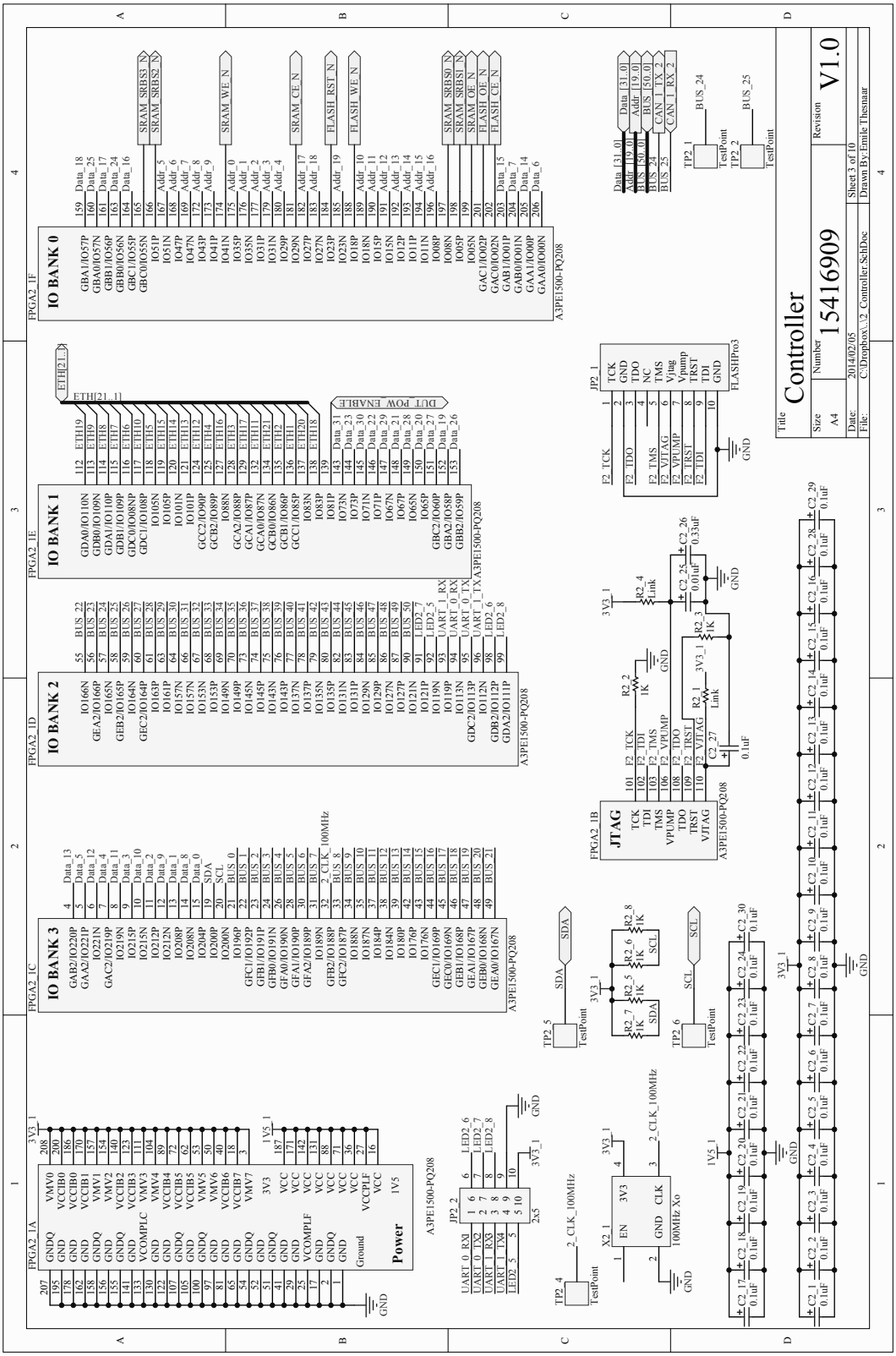

Figure B.3 – Schematic of the Controlling FPGA showing I/O Connections, Crystal, Decoupling Capacitor and Headers

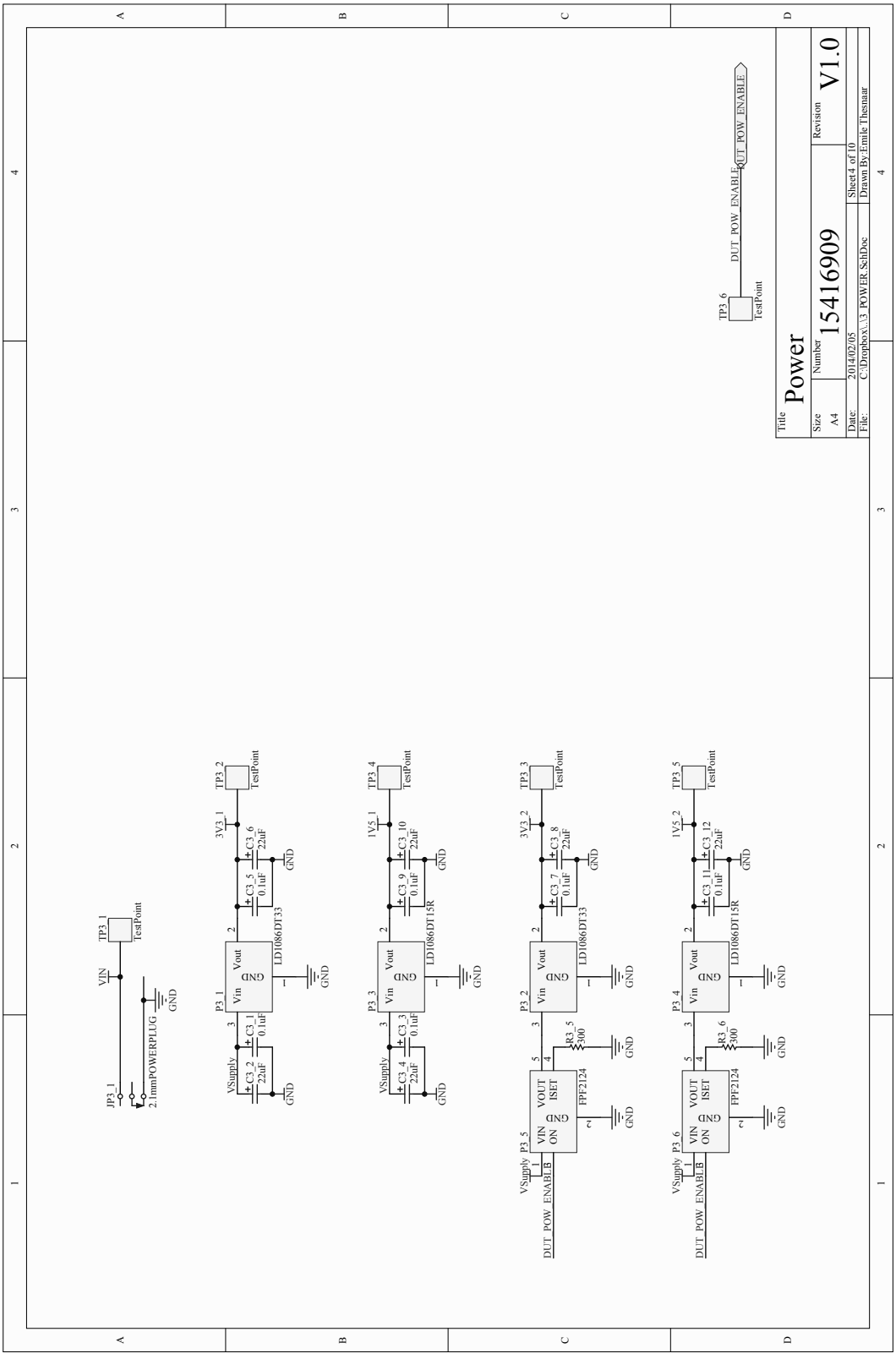

Figure B.4 – Schematic Showing the Connection of the Power Supplies

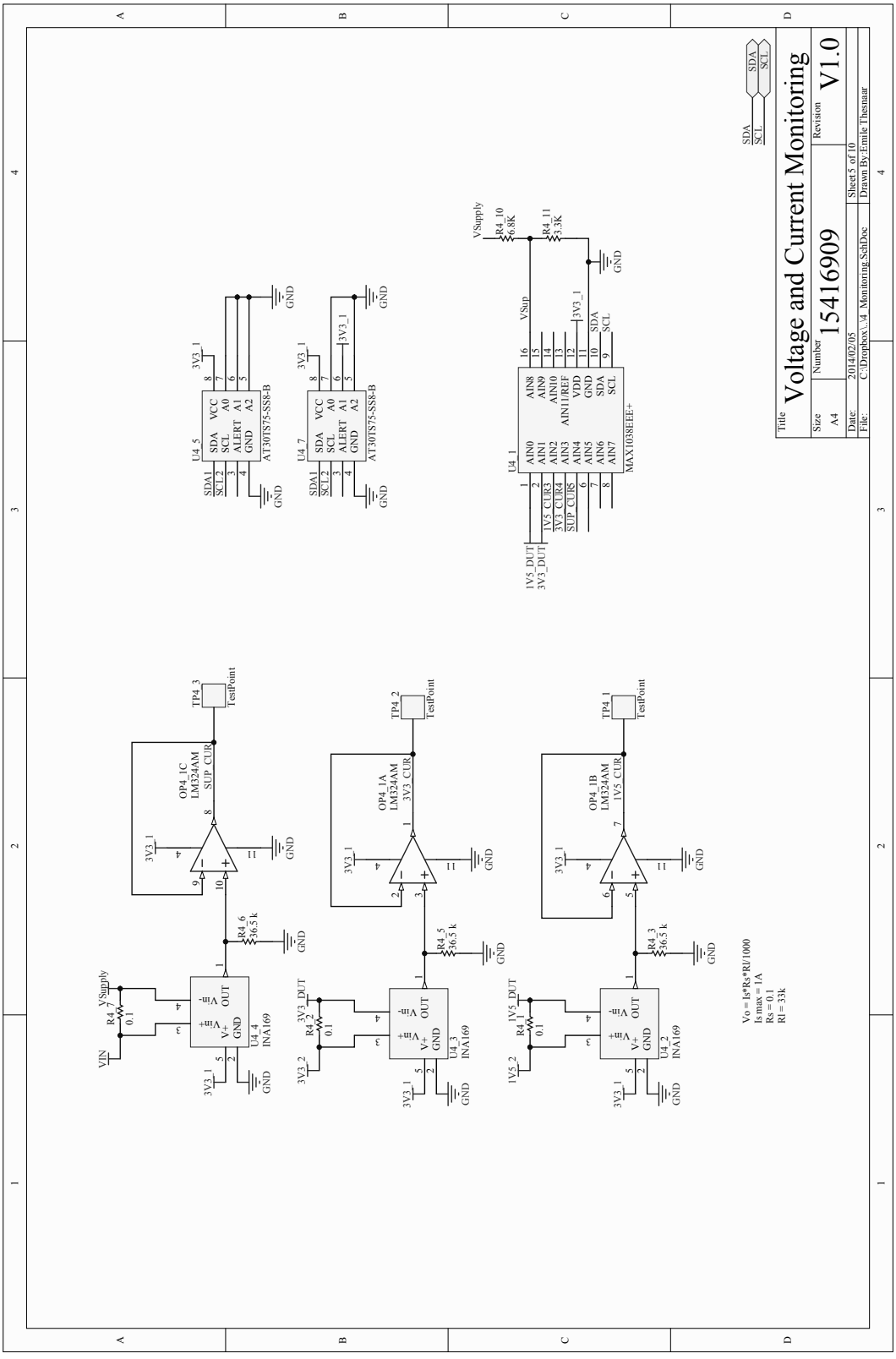

Figure B.5 – Schematic Showing the Current, Voltage and Temperature Sensors

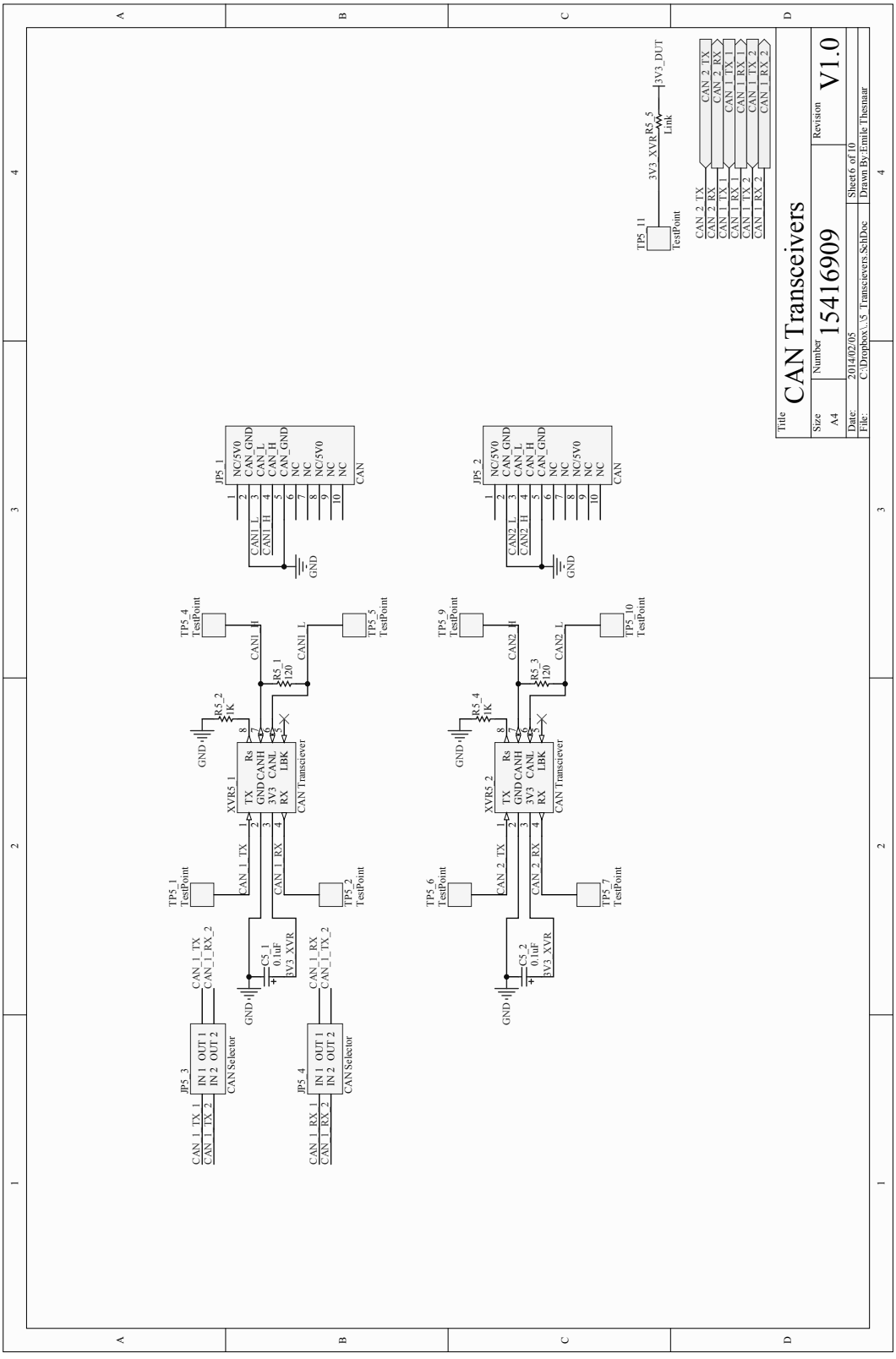

Figure B.6 – Schematic Showing the Connection of the CAN Transceivers

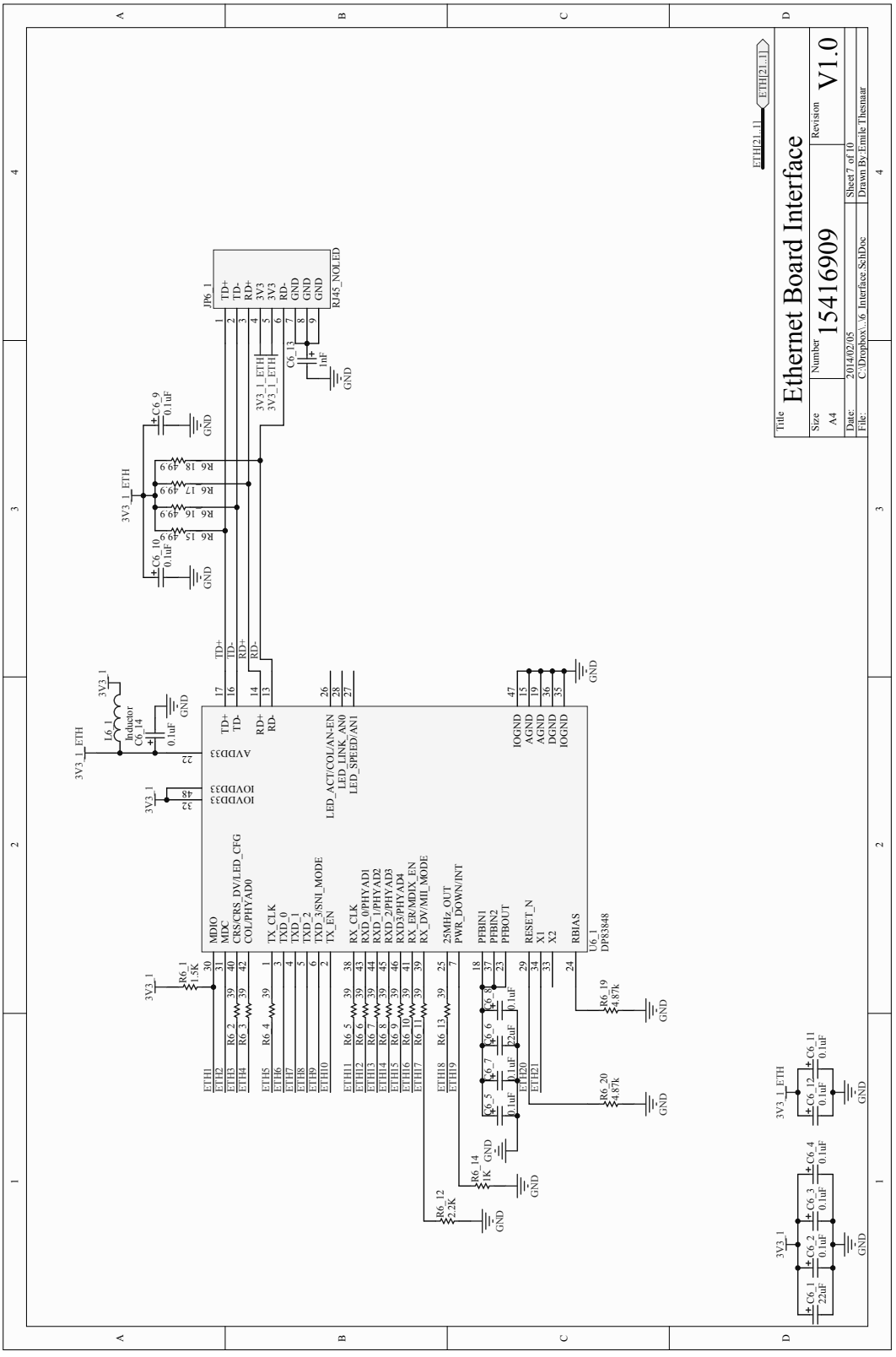

Figure B.7 – Schematic Showing the Connection of the Ethernet PHY

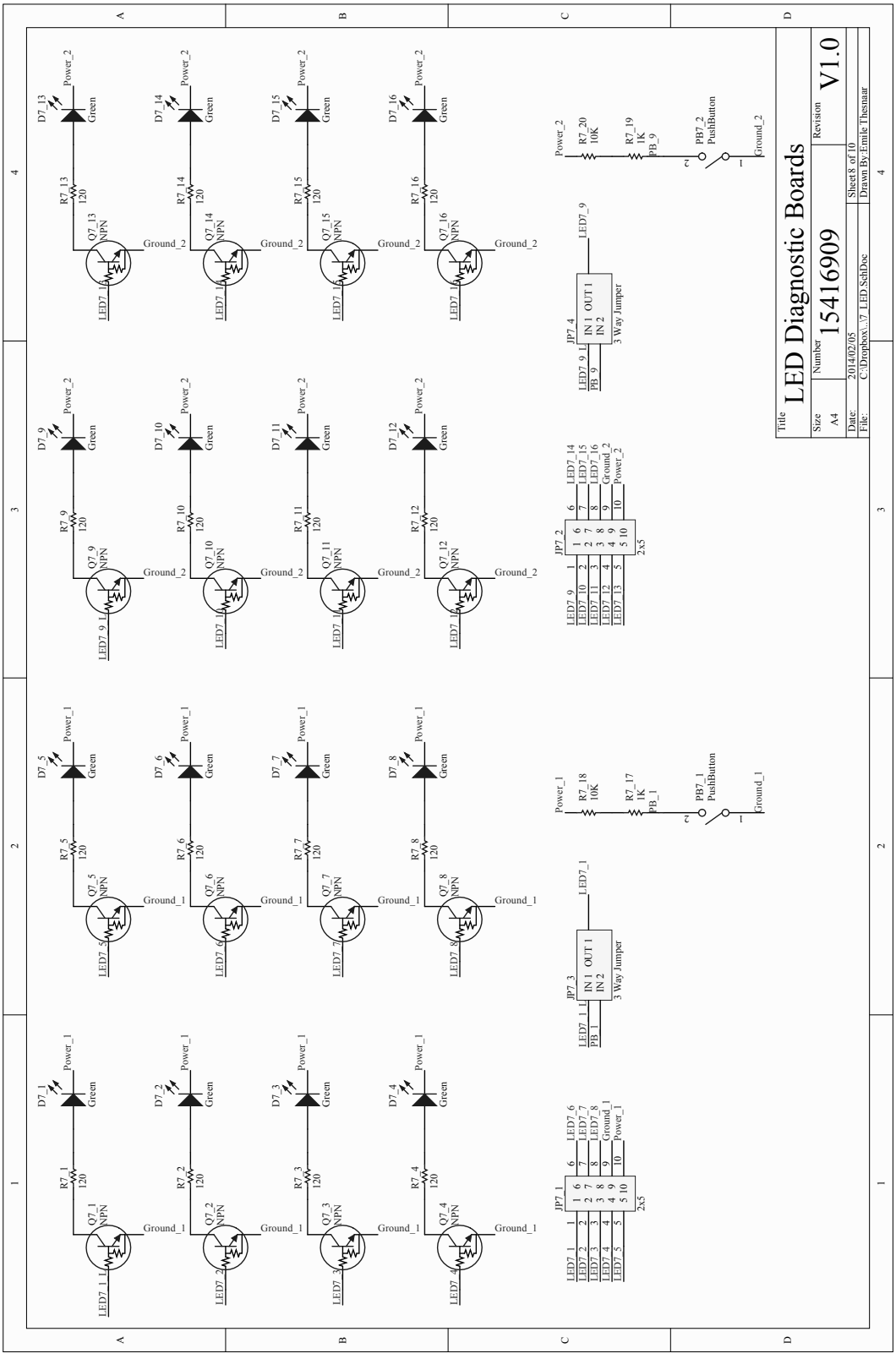

Figure B.8 – Schematic Showing the Connection of the LED Daughter Boards

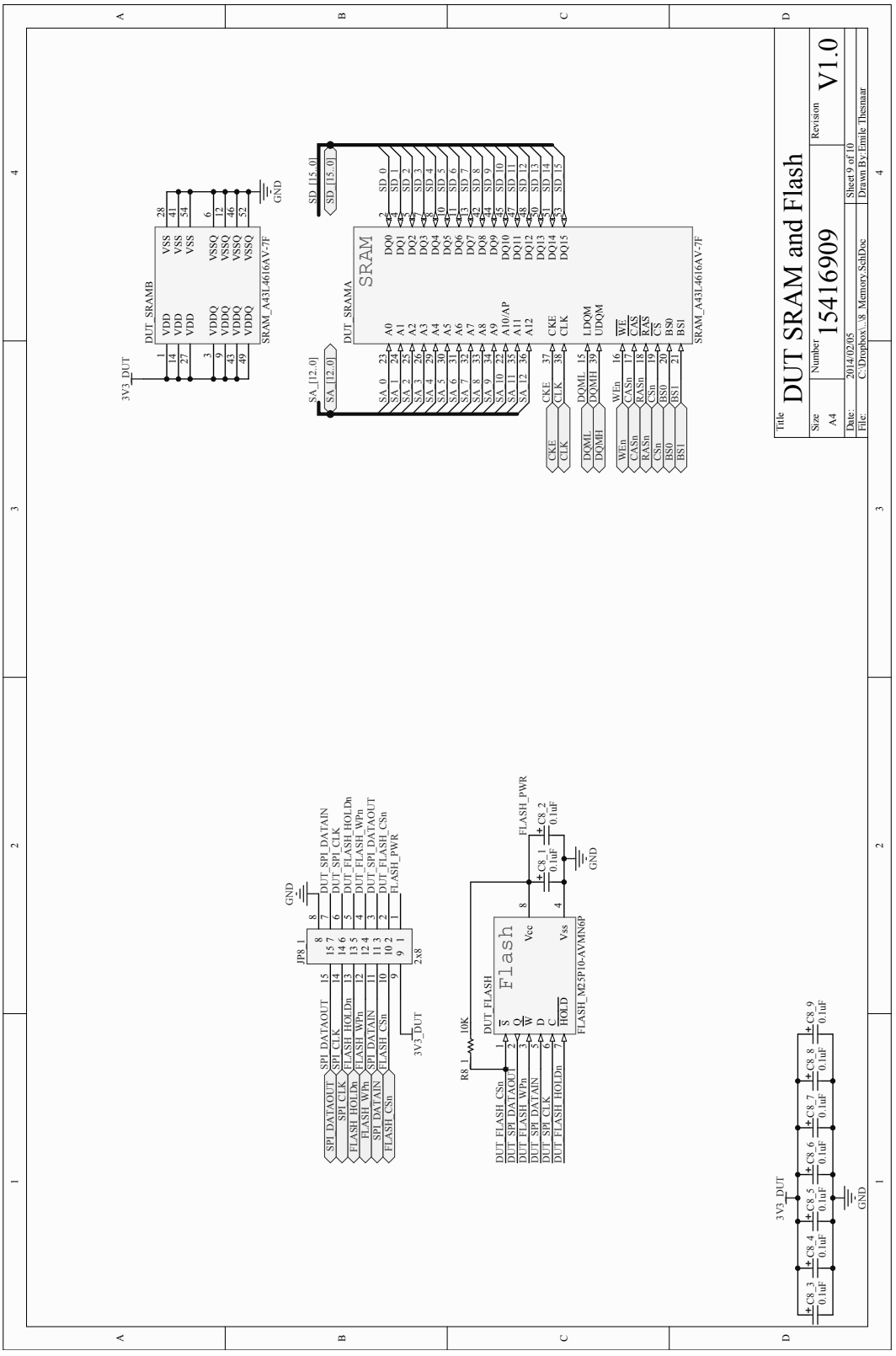

Figure B.9 – Schematic Showing the Connection of the Device Under Test FPGA's Memory

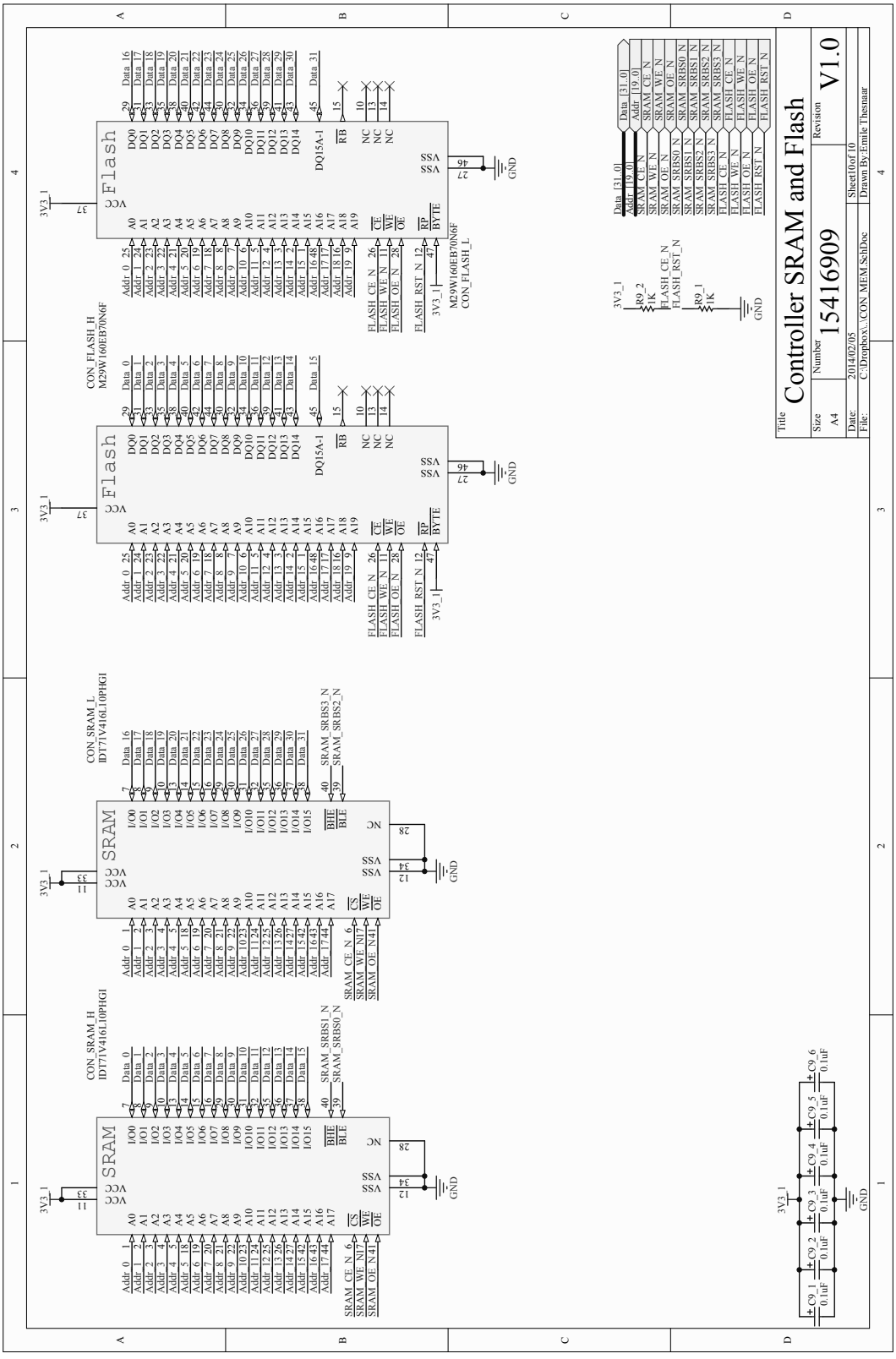

Figure B.10 – Schematic Showing the Connection of the Controlling FPGA's Memory

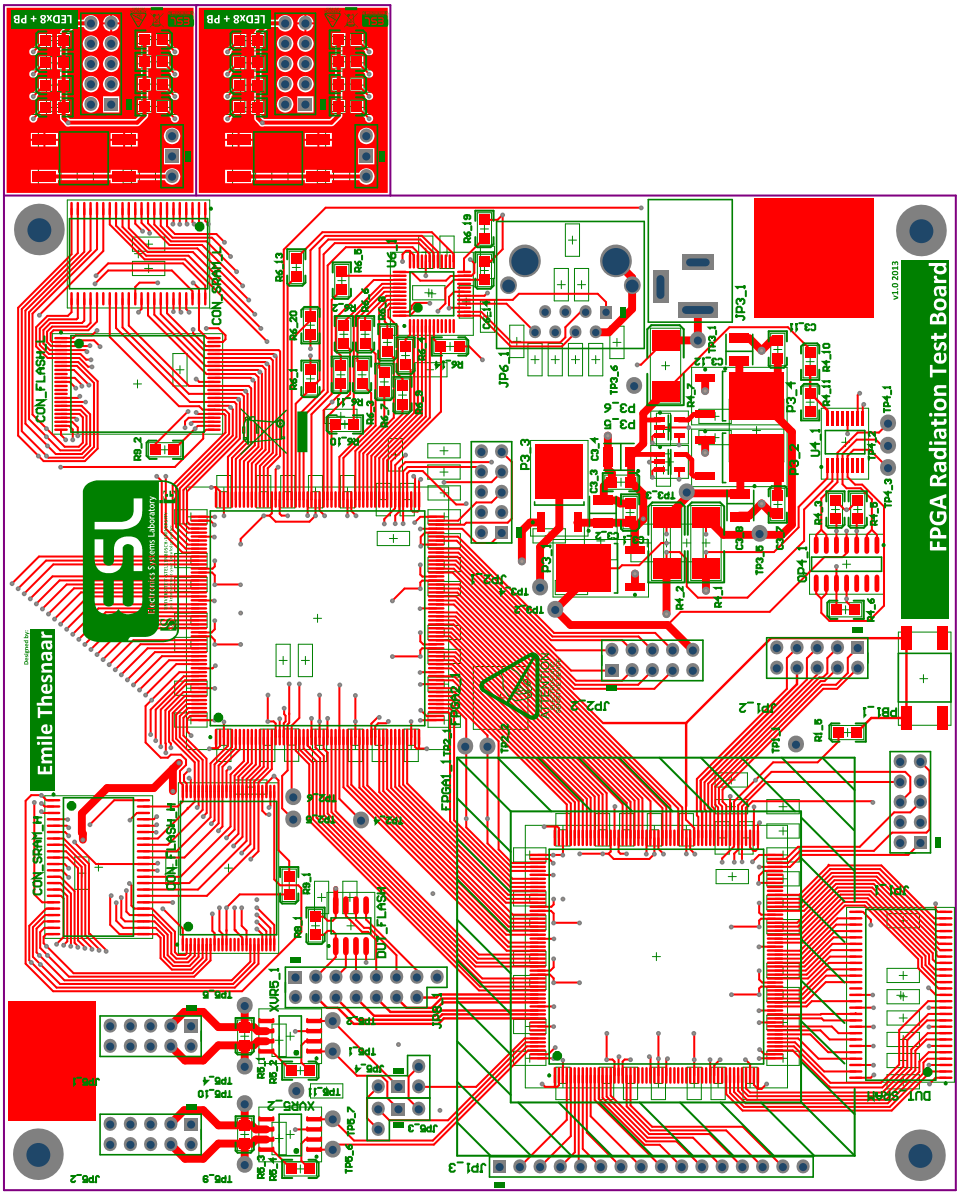

Figure B.11 – Top layer of the PCB, excluding ground plane

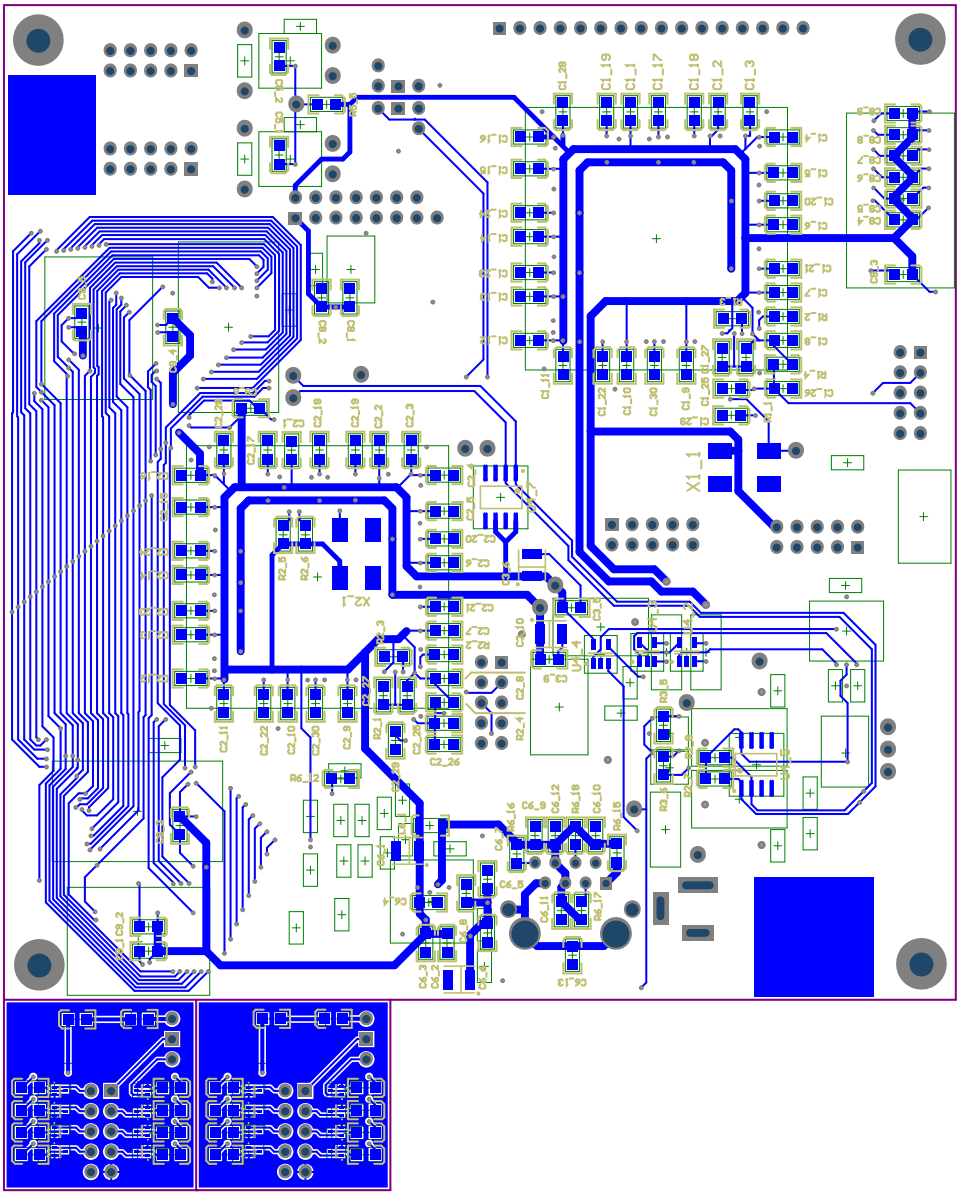

Figure B.12 – Bottom layer of the PCB, excluding ground plane

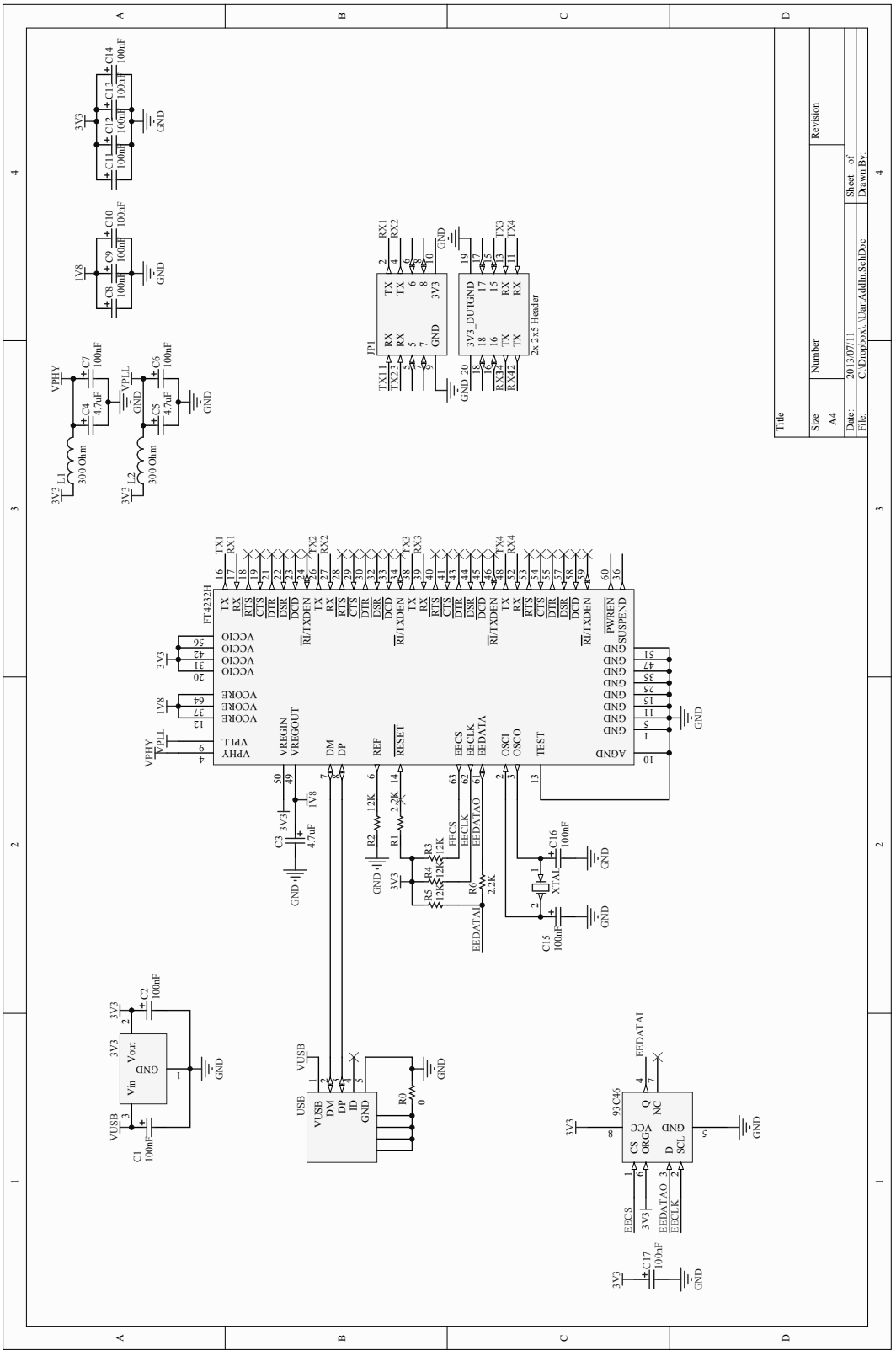

Figure B.13 – Schematic Showing the Connection of the UART Daughter-Board

Appendix C

Source of XML Sprite

```
<?xml version="1.0"?>
<!DOCTYPE flash-device SYSTEM "flash-device.dtd">

<!--//////////////////////////////////////////
// Company: Stellenbosch University: Electronic Systems Laboratory
//
// File: st-m29w160eb-2x8.xml
// File history:
//      v0.9: 22-07-2013: First Model. Working, but not complete
//
// Description:
//
// This file is a template used by the 8051 Debug Tools's Sprite which programs
// the external flash.  Each template is unique to the flash chip used.  This
// one is writen for ST Microelectronics's M29W160EB Flash Chip.
//
// To use this file, copy it to
// <Softconsole Directory>\Sourcery-G++\share\sprite\flash\
// and add the following line to the Memory Map Generator Command
// -M ..\st-m29w160eb-2x8-code-memory.txt
// and place the st-m29w160eb-2x8-code-memory.txt file inside your project
//
// Author: Emile Jacobus Thesnaar
```

```
// Email: 15416909@sun.ac.za (ejt@live.co.za)
//
//////////////////////////////////////////////////-->

<flash-device identifier="st-m29w160eb-2x8">
        <flash-device-version>1.1</flash-device-version>
        <short-name>ST M29W160EB (2x8)</short-name>
        <description>
        The ST M29W160EB flash memory device.

        This flash device description was written to support
        a single flash part connected in 8-bit mode.
        </description>

        <manufacturer-id id="0x0020" />
        <device-id id="0x2249" />

        <!-- Commands go in as single 8-bit words.  -->
        <define-const name="TARGET_WIDTH"
                    value="1"/>

        <!-- Single side in 8-bit mode -->
        <!-- The M29W160EB can be up to 32x64k erase blocks,
            but the 8051
                  is only able to access 64k of it...i.e., about
                      half of one full
                  erase block on the device.  By defining ERASE-
                      ALL-BLOCKS for a
                  single initial command, we can speed things up
                      dramatically by
                  making much smaller individual block sizes
                      used to compare the
                  existing memory contents. -->
        <define-const name="NB_OF_BLOCKS"
                                            value="32" />
        <define-const name="BLOCK_BYTE_SIZE"
                                    value="0x100" />

        <!-- We do 8-bit writes in this version.  -->
        <memory-size blocks-symbol="NB_OF_BLOCKS"
                     size-symbol="BLOCK_BYTE_SIZE"
                     cell-size="0x8"/>
```

```
        <define-const name="READY_DELAY"
                                          value="1"/>

        <!-- Commands -->
        <!-- Reset Command -->
        <define-const name="RESET"
                                                  value="0xF0
            "/>

        <!-- Program Command -->
        <define-const name="PROGRAM_FIRST_ADDRESS"
                                    value="0x800555"/>
        <define-const name="PROGRAM_FIRST_DATA"
                                    value="0xAA"/>
        <define-const name="PROGRAM_SECOND_ADDRESS"
                                    value="0x8002AA"/>
        <define-const name="PROGRAM_SECOND_DATA"
                                    value="0x55"/>
        <define-const name="PROGRAM_THIRD_ADDRESS"
                                    value="0x800555"/>
        <define-const name="PROGRAM_THIRD_DATA"
                                    value="0xA0"/>

        <!-- Erase Command -->
        <define-const name="ERASE_FIRST_ADDRESS"
                                    value="0x800555"/>
        <define-const name="ERASE_FIRST_DATA"
                                    value="0xAA"/>
        <define-const name="ERASE_SECOND_ADDRESS"
                                    value="0x8002AA"/>
        <define-const name="ERASE_SECOND_DATA"
                                    value="0x55"/>
        <define-const name="ERASE_THIRD_ADDRESS"
                                    value="0x800555"/>
        <define-const name="ERASE_THIRD_DATA"
                                    value="0x80"/>
        <define-const name="ERASE_FOURTH_ADDRESS"
                                    value="0x800555"/>
        <define-const name="ERASE_FOURTH_DATA"
                                    value="0xAA"/>
```

```
        <define-const name="ERASE_FIFTH_ADDRESS"
                                  value="0x8002AA"/>
        <define-const name="ERASE_FIFTH_DATA"
                                  value="0x55"/>
        <define-const name="ERASE_SIXTH_ADDRESS"
                                  value="0x800555"/>
        <define-const name="ERASE_SIXTH_DATA"
                                  value="0x10"/>
        <define-const name="ERASE_ADDRESS"
                                        value="0x800000"/>
        <define-const name="ERASE_SUCCESS"
                                        value="0xFF"/>
        <define-const name="ERASE_DELAY"
                                        value="48000"/>


        <!-- <program-init/> -->

        <program>
                <!-- Look to be given three arguments.  -->
                <keep-arg name="ADDRESS"/>
                <keep-arg name="DATA"/>
                <keep-arg name="SIZE"/>

                <local-var name="NB_BYTES"/>
                <until counter-name="NB_BYTES" limit-symbol="
                   SIZE">

                     <!-- Write Program command and data -->
                     <write-value width-symbol="TARGET_WIDTH
                        " address-symbol="
                        PROGRAM_FIRST_ADDRESS" value-symbol
                        ="PROGRAM_FIRST_DATA"/>
                     <write-value width-symbol="TARGET_WIDTH
                        " address-symbol="
                        PROGRAM_SECOND_ADDRESS" value-
                        symbol="PROGRAM_SECOND_DATA"/>
                     <write-value width-symbol="TARGET_WIDTH
                        " address-symbol="
                        PROGRAM_THIRD_ADDRESS" value-symbol
                        ="PROGRAM_THIRD_DATA"/>
```

```
                        <write-data address-symbol="ADDRESS"
                            data-addr-symbol="DATA" width="1"/>

                        <!-- Do some checking here -->

                <!-- We do a single byte at a time.  -->
                        <incr name="ADDRESS" value="1"/>
                        <incr name="DATA" value="1"/>
                        <incr name="NB_BYTES" value="1"/>
                </until>

        </program>

        <!-- <program-fini/> -->

        <!-- <block-erase/> -->

        <erase-all-blocks>
                <!-- Write chip erase command -->
                <write-value width-symbol="TARGET_WIDTH"
                    address-symbol="ERASE_FIRST_ADDRESS" value-
                    symbol="ERASE_FIRST_DATA"/>

                <write-value width-symbol="TARGET_WIDTH"
                    address-symbol="ERASE_SECOND_ADDRESS" value
                    -symbol="ERASE_SECOND_DATA"/>

                <write-value width-symbol="TARGET_WIDTH"
                    address-symbol="ERASE_THIRD_ADDRESS" value-
                    symbol="ERASE_THIRD_DATA"/>

                <write-value width-symbol="TARGET_WIDTH"
                    address-symbol="ERASE_FOURTH_ADDRESS" value
                    -symbol="ERASE_FOURTH_DATA"/>

                <write-value width-symbol="TARGET_WIDTH"
                    address-symbol="ERASE_FIFTH_ADDRESS" value-
                    symbol="ERASE_FIFTH_DATA"/>

                <write-value width-symbol="TARGET_WIDTH"
                    address-symbol="ERASE_SIXTH_ADDRESS" value-
                    symbol="ERASE_SIXTH_DATA"/>
```

```
                <check-status>
                        <!-- Wait till Finish -->
                        <local-var name="CHECK" init="0x00"/>
                        <wait-for address-symbol="ERASE_ADDRESS
                            " timeout-symbol="ERASE_DELAY">
                                <expect-bit value="
                                    ERASE_SUCCESS" width-symbol
                                    ="TARGET_WIDTH"/>
                        </wait-for>
                </check-status>

        </erase-all-blocks>

        <!-- <reset/> -->

        <!-- <unlock-block/> -->

</flash-device>
```

Listing C.1 – Sprite for SoftConsole to program memory

Appendix D

Libero Design

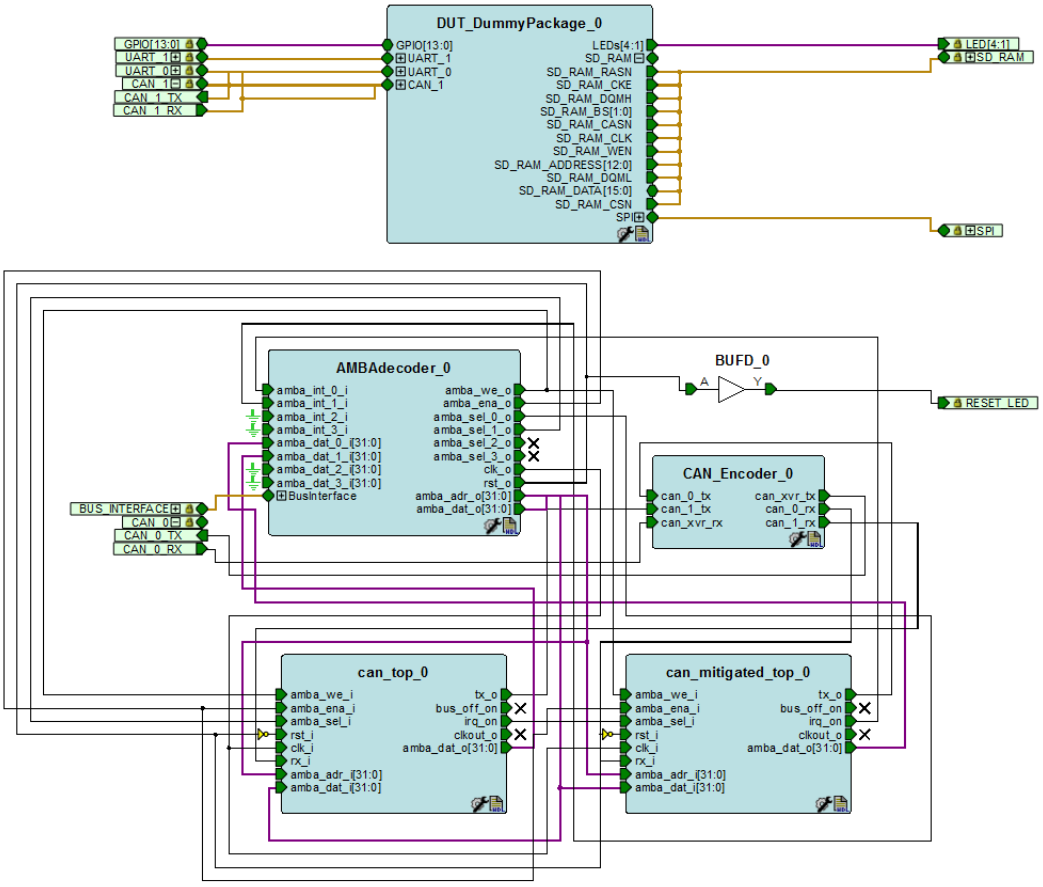

Figure D.1 – Final Libero Design for the Device Under Test

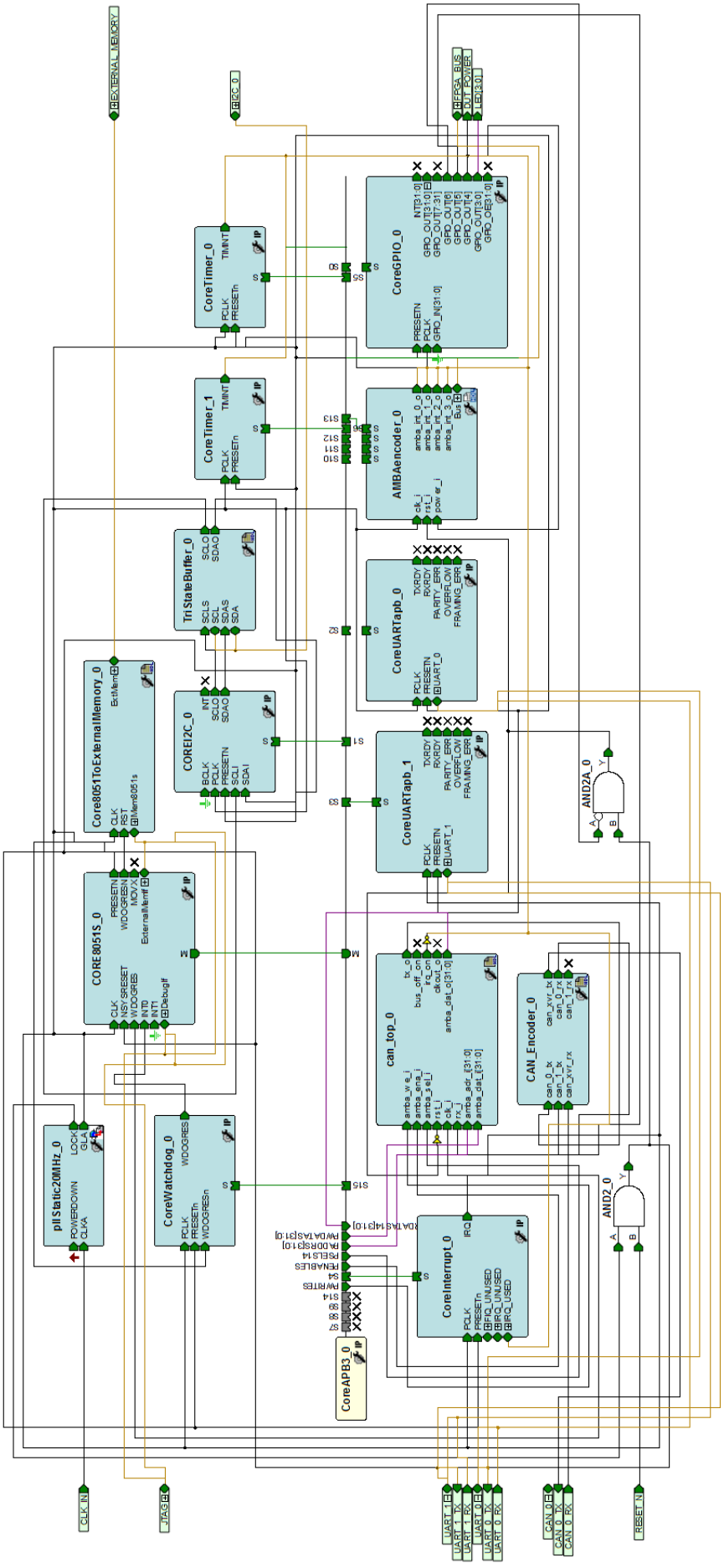

Figure D.2 – Final Libero Design for the Controller

Appendix E

Simulation Waveforms

This page has been left blank intentionally.

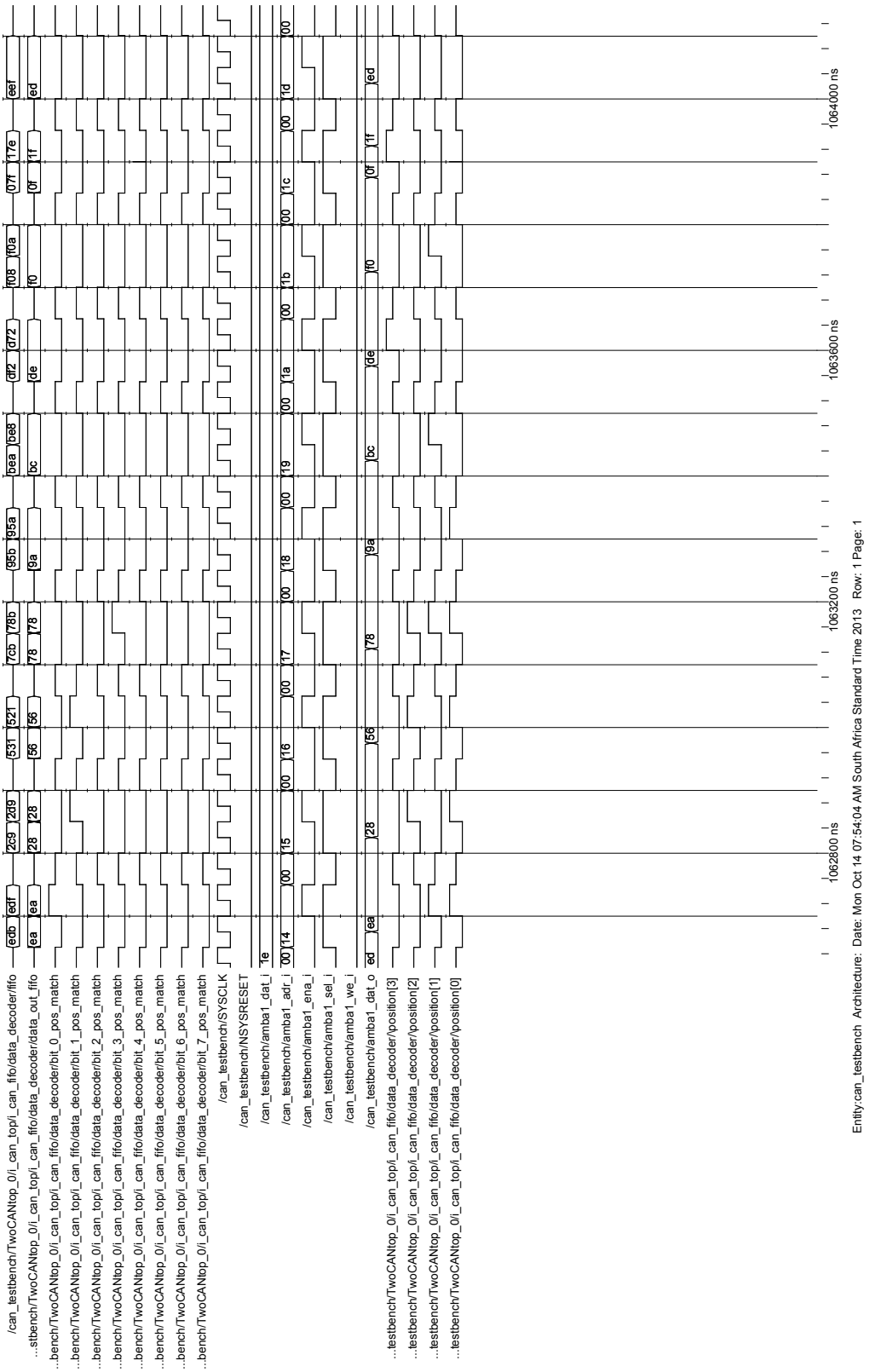

Figure E.1 – Timing Diagram of the Mitigated CAN Controller's FIFO Module With Injected Errors Showing Mitigation and Hamming Code Recovery

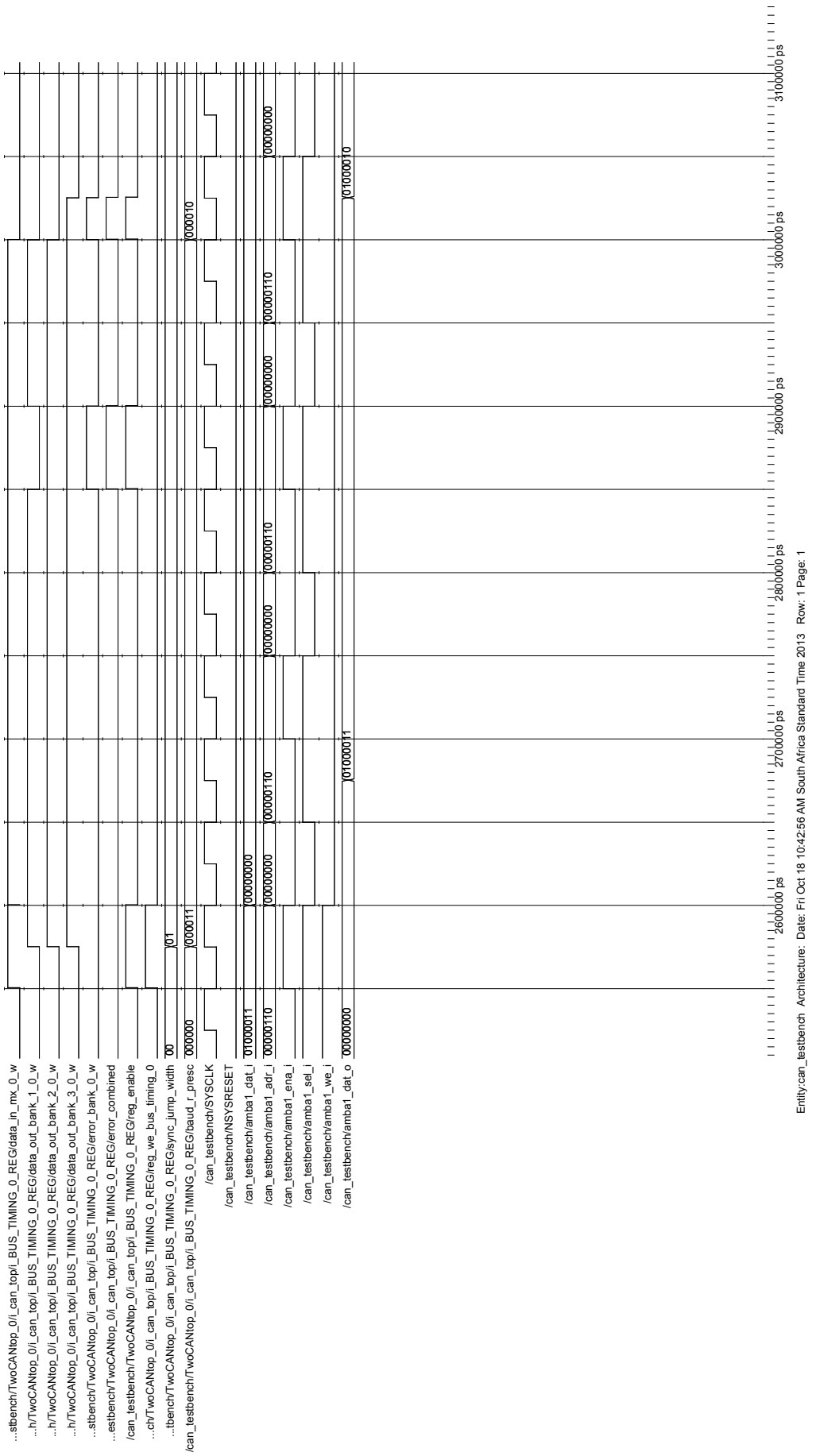

Figure E.2 – Timing Diagram of the Mitigated CAN Controller's Bus Timing 0 Register With Injected Errors Showing Mitigation and TMR Recovery

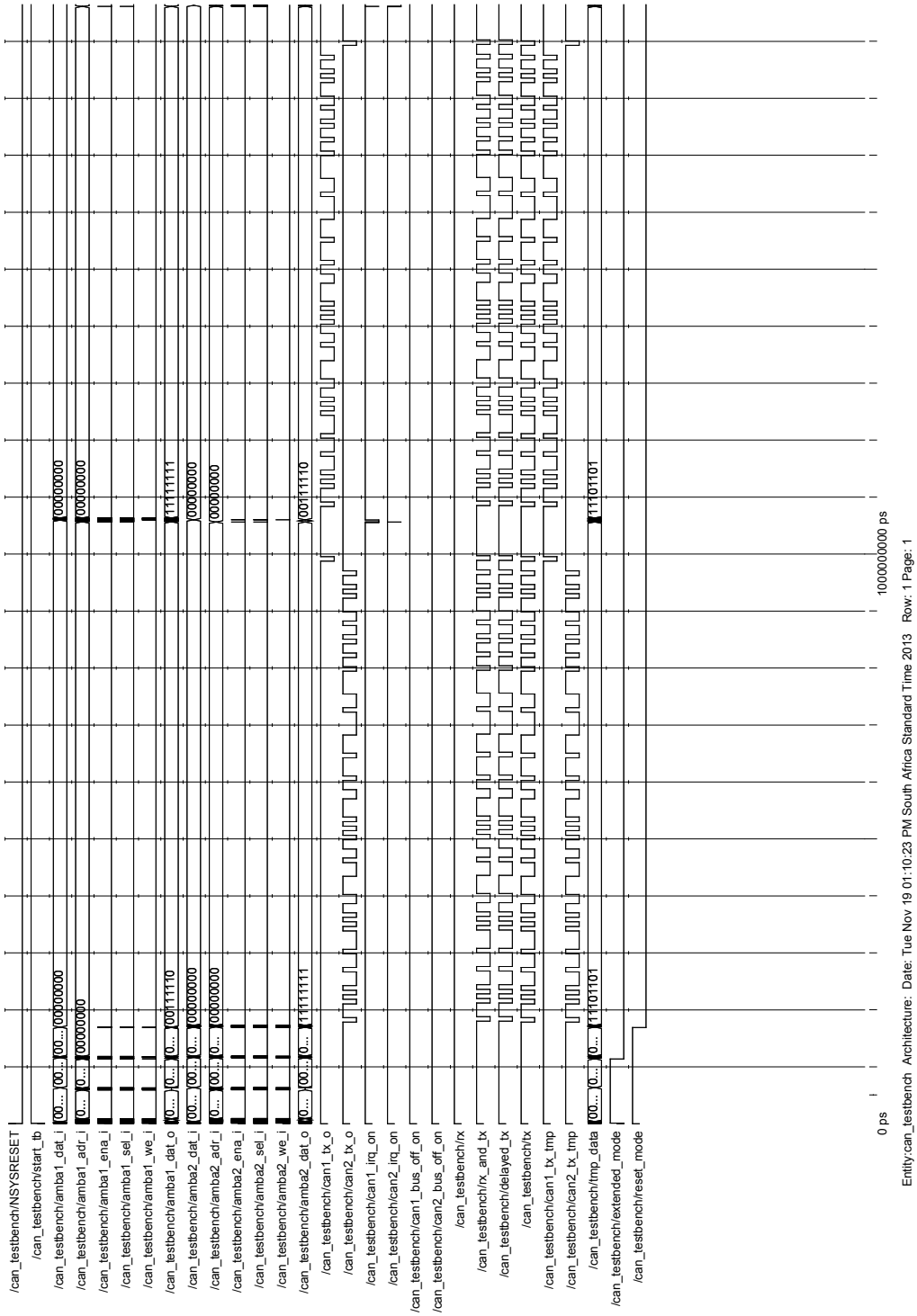

Figure E.3 – Simulation showing message transmission and reception between the mitigated and unmitigated CAN controller

Appendix F

iThemba Test Log

Time	Message
30/10/13 15:00:00	Beam defocussing started, Beam at 1nA
30/10/13 15:00:02	Beam on
Prelimanary Test 1	
30/10/13 16:31:20	Test Started, Beam at 1nA, Board at 90^o
30/10/13 16:33:20	Test Stopped
Prelimanary Test 2	
30/10/13 16:35:02	Test Started, Beam at 1nA, Board at 90^o
30/10/13 16:37:02	Test Stopped
31/10/13 16:37:02	1.5V supply to DUT at 11mA
31/10/13 16:37:02	3.3V supply to DUT at 27mA
31/10/13 16:37:02	5.25V supply to Board at 190mA
31/10/13 16:37:02	PCB temp between FPGAs at 29.75 ^{o}C
31/10/13 16:37:02	PCB temp at PSUs at 35.06 ^{o}C
30/10/13 16:41:02	Beam off
30/10/13 16:41:02	Beam defocussing complete
30/10/13 16:41:02	Power Cycle
30/10/13 16:47:00	Moving DUT to 0^o
Test 1	
30/10/13 16:48:02	Test Started, Beam turned on at 0.1nA, Board at 0^o
30/10/13 16:48:52	Test Aborted due to no particles being measured. This was due to particle current being to small to measure. DUT was in Beam for 27 seconds.
30/10/13 16:48:52	Test Stopped, Beam off
31/10/13 16:48:52	Core current to DUT at 12mA, rest normal
Test 2	
30/10/13 16:52:34	Test Started, Beam turned on at 0.1nA, Board at 0^o
30/10/13 16:54:34	Test Stopped, Beam off
31/10/13 16:54:34	Core current to DUT at 11mA, rest normal

Table F.1 – iThemba Test Logs Part 1

Test 3	
30/10/13 16:56:00	Test Started, Beam turned on at 0.1nA, Board at 0^o
30/10/13 17:03:00	Several errors on all CAN controllers noted.
30/10/13 16:58:00	Test Stopped, Beam off
30/10/13 16:59:00	Power Cycle
31/10/13 17:59:00	Core current to DUT at 12mA, rest normal
Test 4	
30/10/13 17:00:59	Test Started, Beam turned on at 0.1nA, Board at 0^o
30/10/13 17:03:00	Several errors on all CAN controllers noted.
30/10/13 17:03:00	Test Stopped, Beam off
31/10/13 17:03:00	Core current to DUT at 22mA, rest normal
Test 5	
30/10/13 17:06:30	Test Started, Beam turned on at 0.1nA, Board at 0^o
30/10/13 17:08:30	Test Stopped, Beam off
31/10/13 17:08:30	Core current to DUT at 22mA, rest normal
Test 6	
30/10/13 17:11:30	Test Started, Beam turned on at 0.2nA, Board at 0^o
30/10/13 17:13:30	Failure in CAN Controller 1 config
30/10/13 17:13:30	Test Stopped, Beam off
31/10/13 17:13:02	1.5V supply to DUT at 26mA
31/10/13 17:13:02	3.3V supply to DUT at 27mA
31/10/13 17:13:02	5.23V supply to Board at 245mA
31/10/13 17:13:02	PCB temp between FPGAs at 29.63 ^{o}C
31/10/13 17:13:02	PCB temp at PSUs at 36.81 ^{o}C
End of testing. Francois turn	
30/10/13 17:16:00	Reprogramming for Francois
30/10/13 17:16:30	Erase Failed, Suspect Cabling or FPGA Damage
Annealing 30/10/13 17:16:30 - 31/10/13 09:31:00	
31/10/13 09:31:30	Further attempts at reprogramming failed, even after opening of Tank.
31/10/13 09:31:30	Damage to Charge Pumps suspected.
31/10/13 09:31:30	Core current to DUT at 22mA, rest normal
Post Failure Tests 1	
31/10/13 09:36:40	Test Started, no vaccum and tank open
31/10/13 09:36:50	Failure in all CAN 2 config registers.
31/10/13 09:36:50	Partial erase prior to charge pump failure expected.
31/10/13 09:38:40	Test Stopped
31/10/13 09:38:30	Core current to DUT at 22mA, rest normal
Post Failure Tests 2	
31/10/13 09:39:30	Test Started, no vaccum and tank open
31/10/13 09:39:40	Failure in all CAN 2 config registers.
31/10/13 09:41:30	Test Stopped
31/10/13 09:41:45	Power Cycle
31/10/13 09:41:30	Core current to DUT at 22mA, rest normal
Post Failure Tests 3	
31/10/13 09:43:30	Test Started, no vaccum and tank open
31/10/13 09:43:40	Failure in all CAN 2 config registers.
31/10/13 09:45:30	Test Stopped
31/10/13 09:45:30	Core current to DUT at 22mA, rest normal

Table F.2 – iThemba Test Logs Part 2